2INC0 - Operating Systems

Introduction to Operating Systems

Dr. Geoffrey Nelissen

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

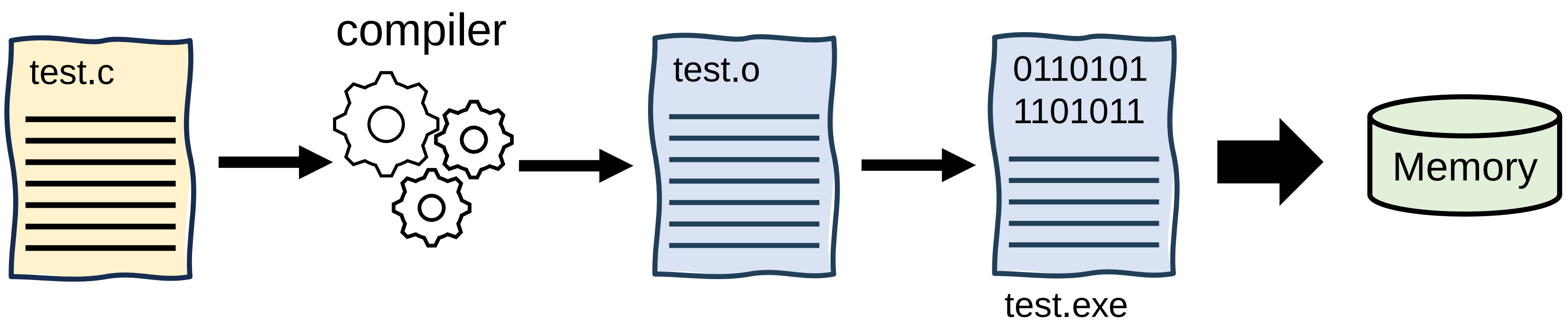
Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lecture 2)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 3)
  - action synchronization (lecture 4)
  - condition synchronization (lecture 5)
  - deadlock (lecture 6)
- **Memory management** (lecture 7 ad 8)
- **Input/output** (lectures 9 and 10)
  - general issues
  - file systems

Agenda

- Course overview
- OS: place in the computer system
- Motivation and OS tasks
- Extra-functional requirements

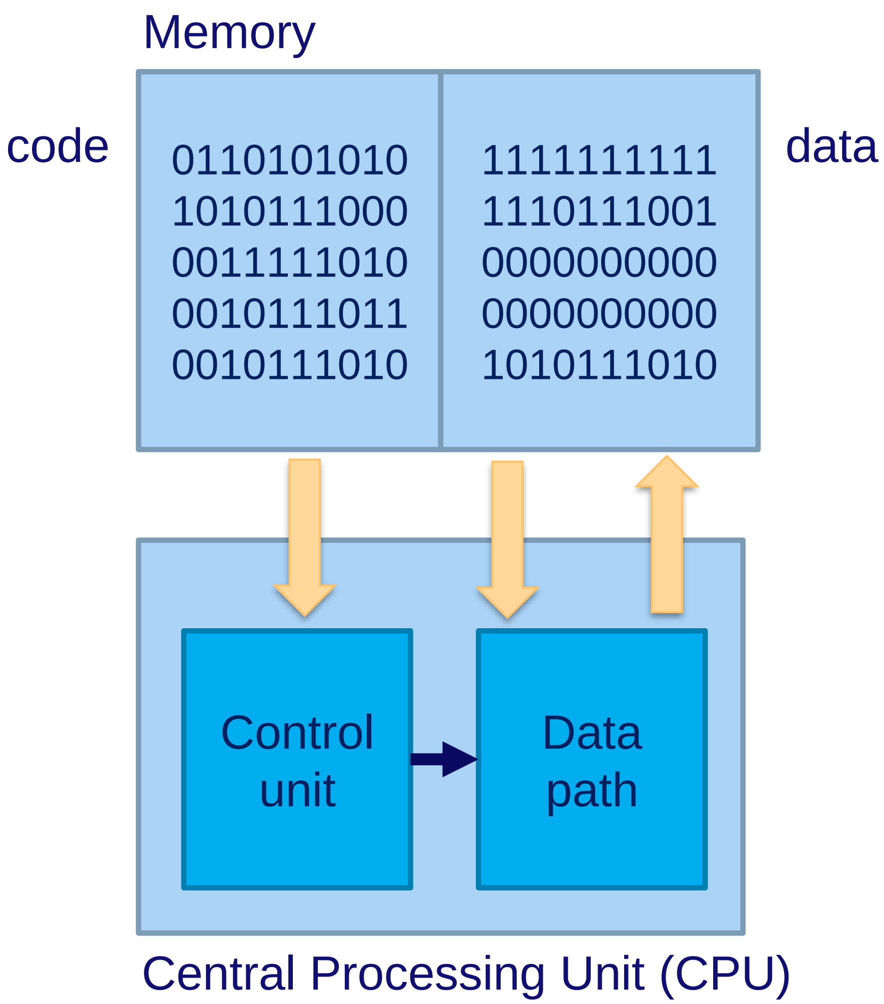
What are the steps to execute a program?



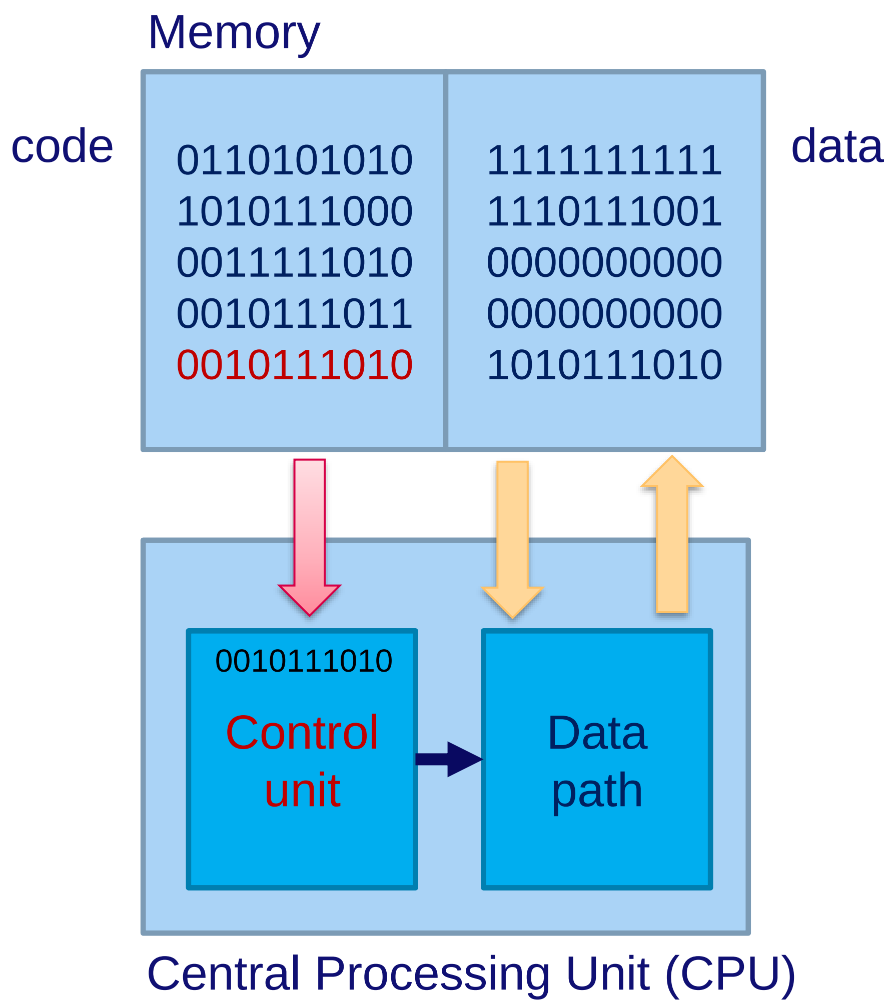
Executable file is **platform dependent** (ISA, cpu arch, …)

How is code executed by your computer?

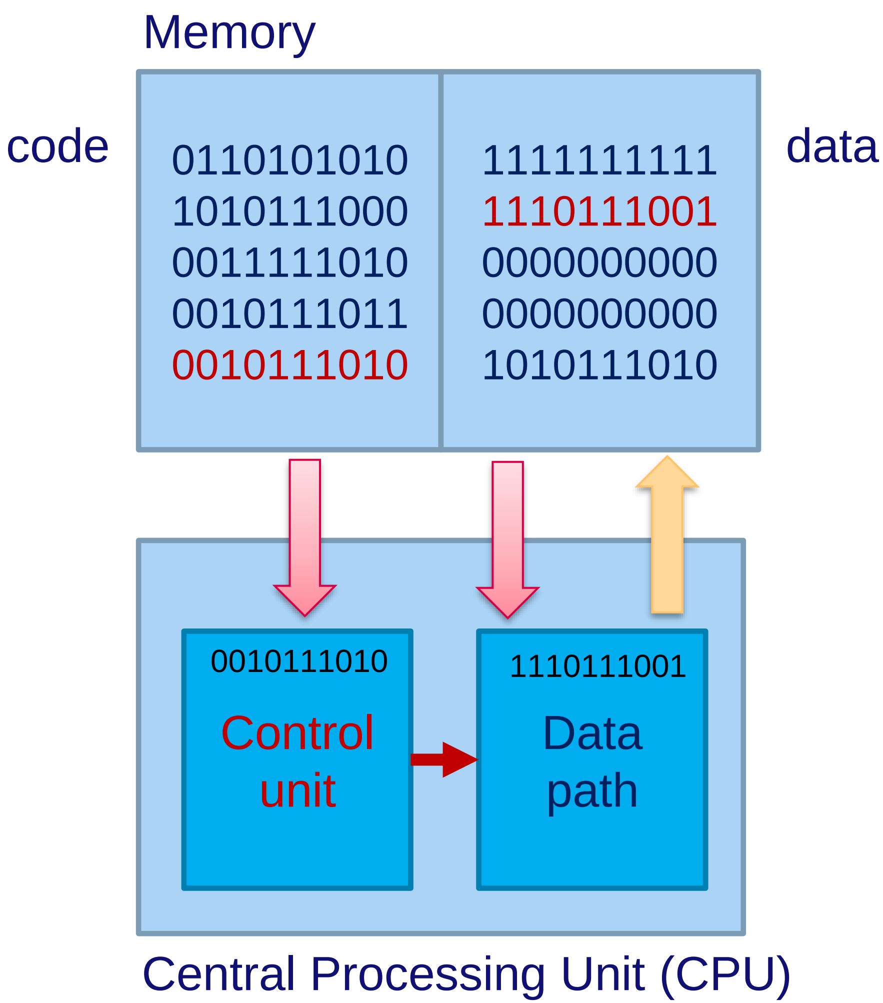
How is code executed by your computer?



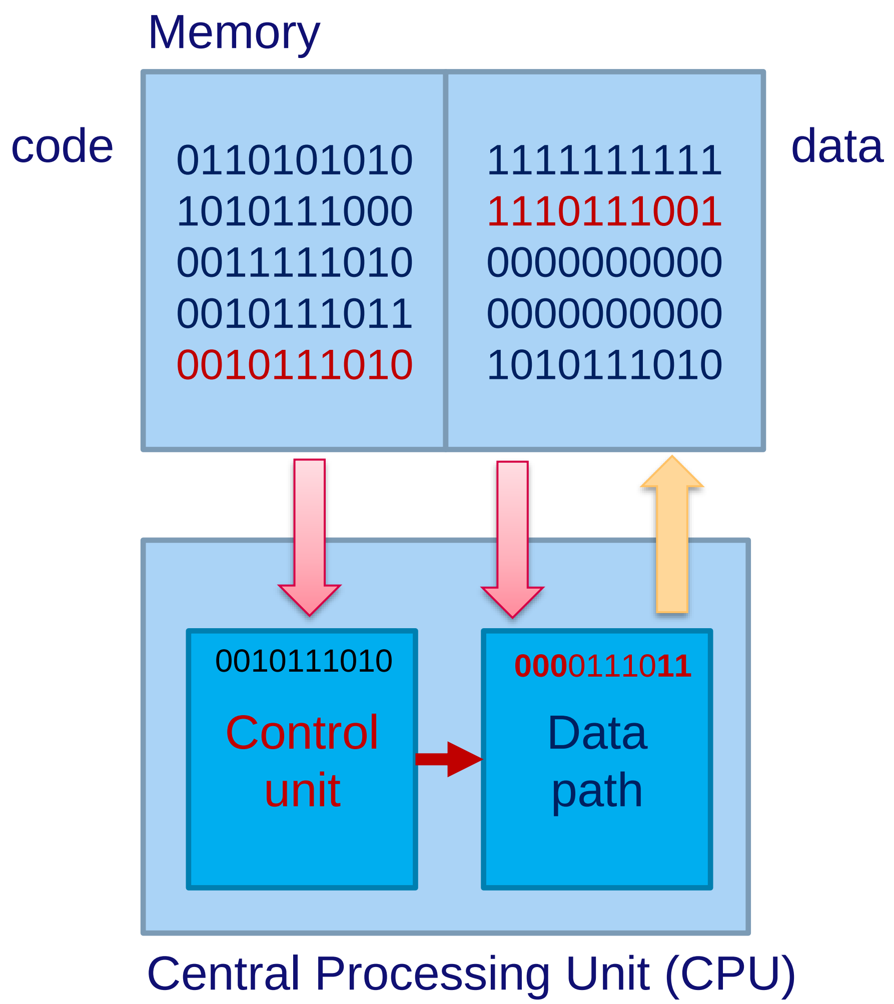
How is code executed by your computer?



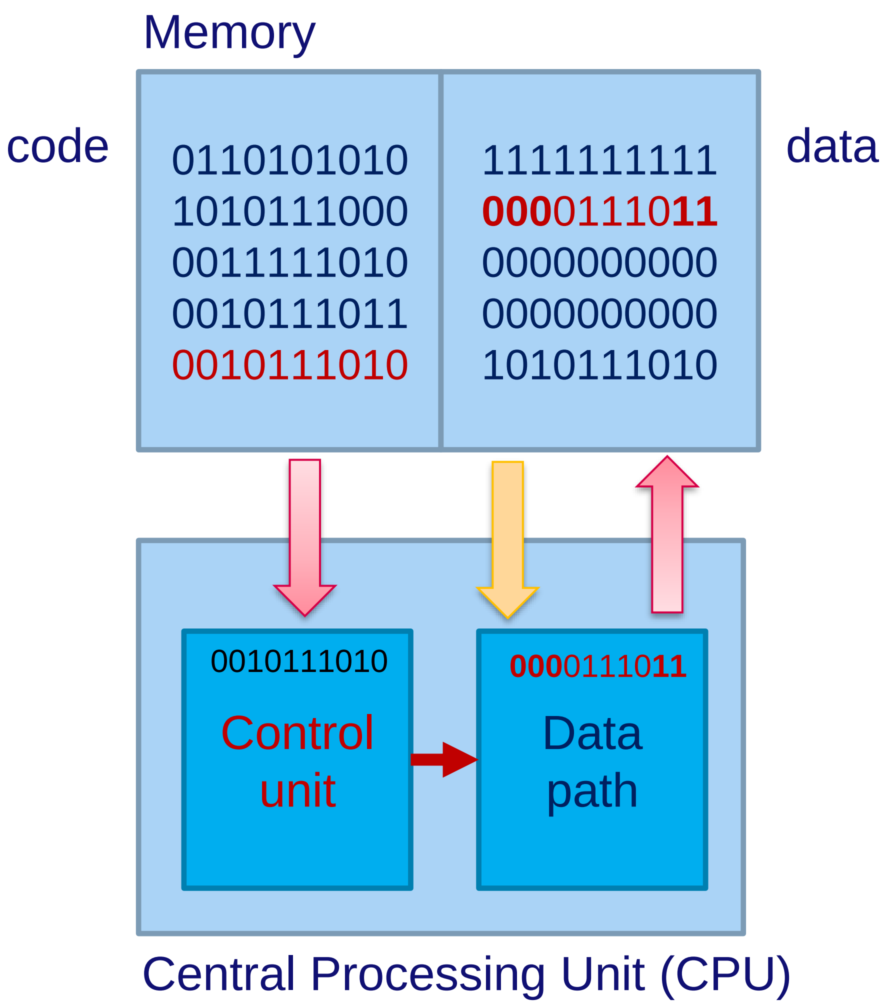
How is code executed by your computer?



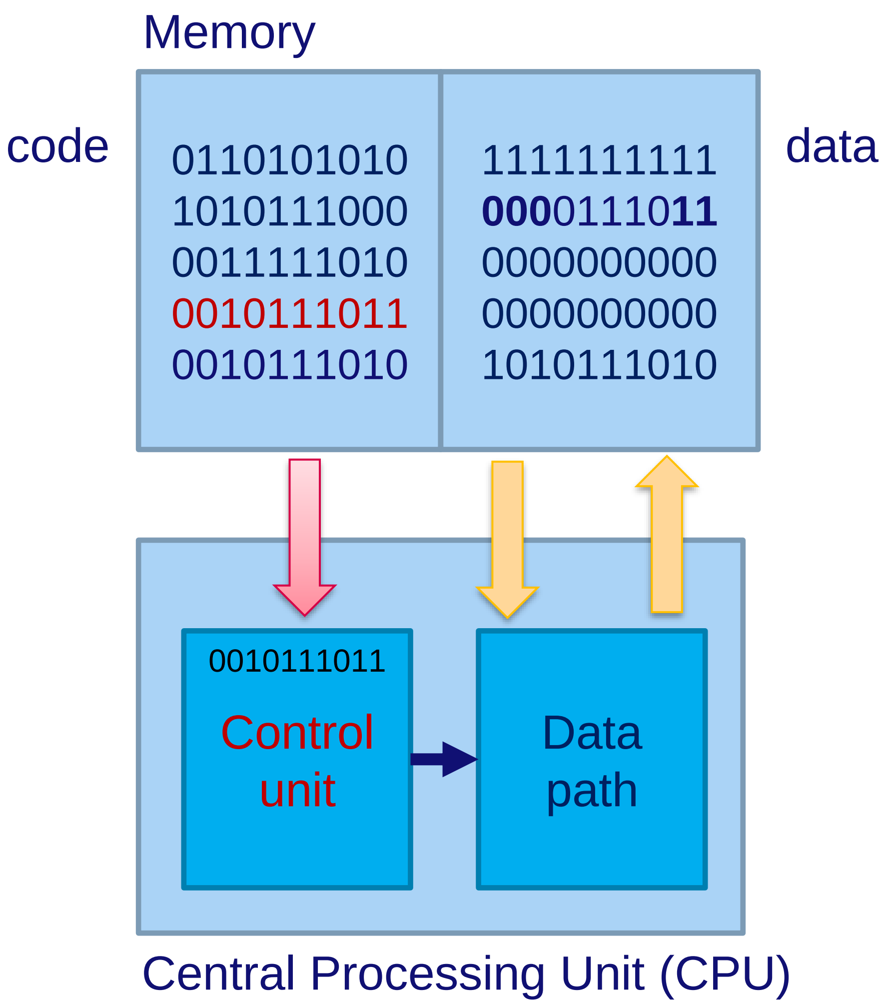
How is code executed by your computer?



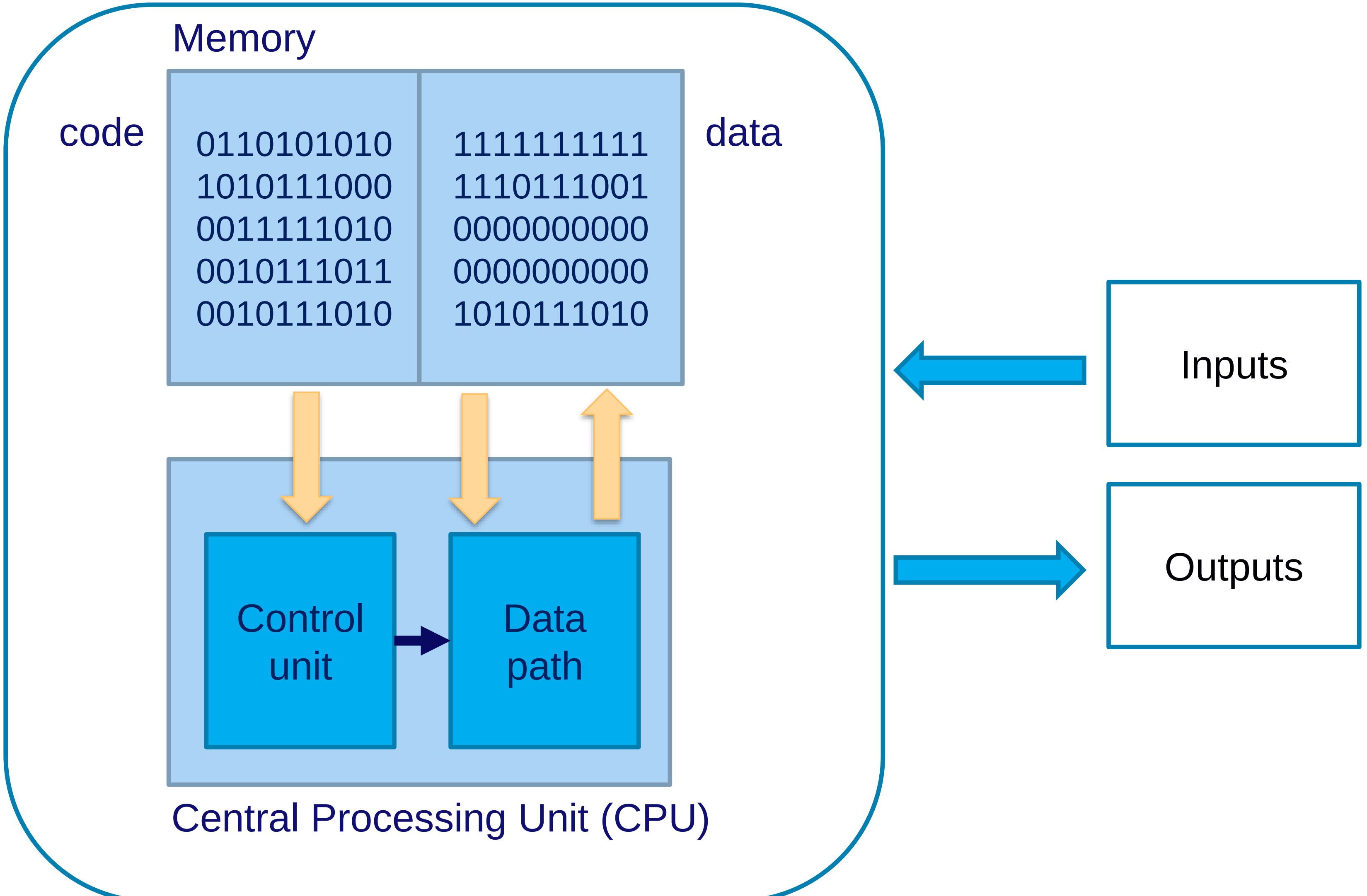
How is code executed by your computer?



How is code executed by your computer?

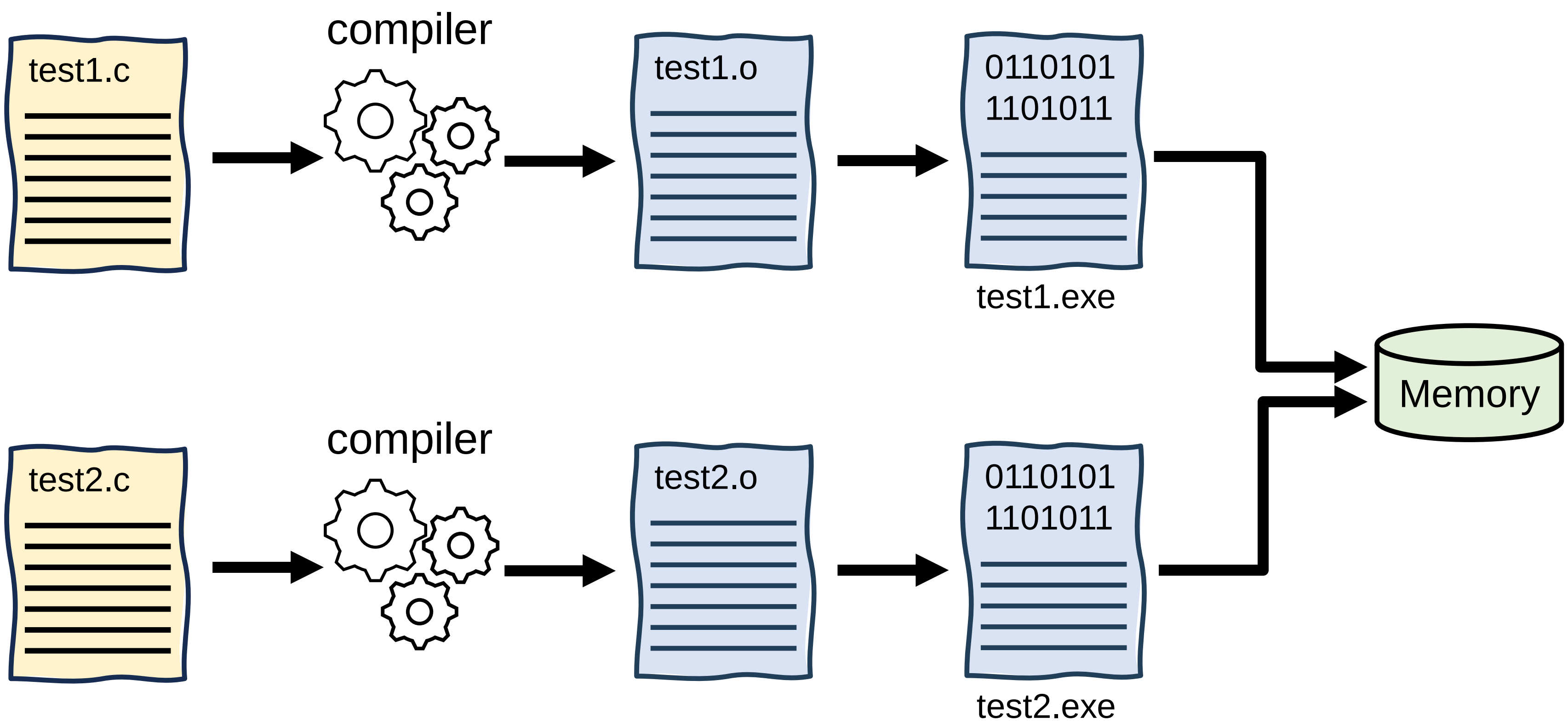


How is code executed by your computer?



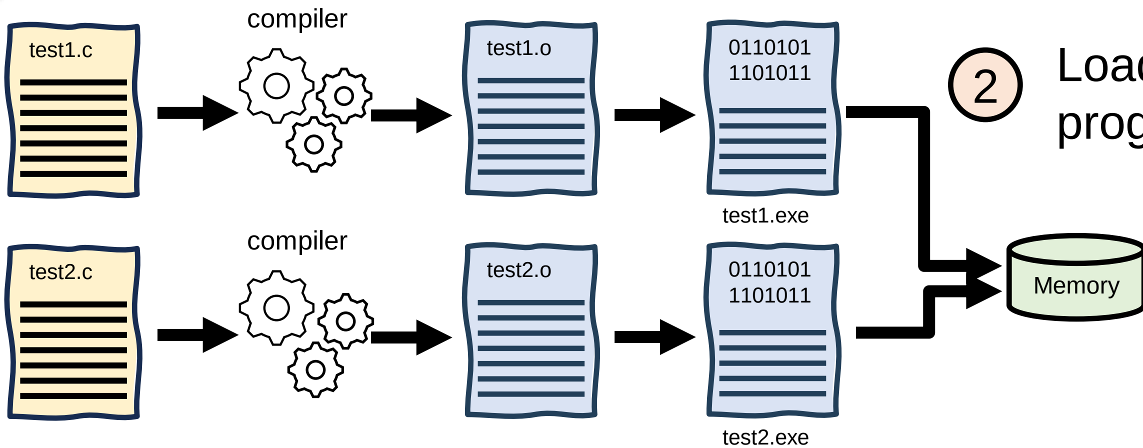
Why do we need an operating system?

What are the steps to execute a program?

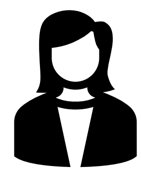


What executes?

What are the steps to execute a program?



2 Load additional programs in memory



Main program that always runs on the processor from system startup and **manages execution of other programs**

3 Request to execute test1.exe (using system call)

1 When system starts, start executing the operating system

Operating System

4 Operating system starts execution of test1.exe

Processor

Any other reason we might want to use an operating system?

Elements of a computer system

Users

- humans
- machines

Software

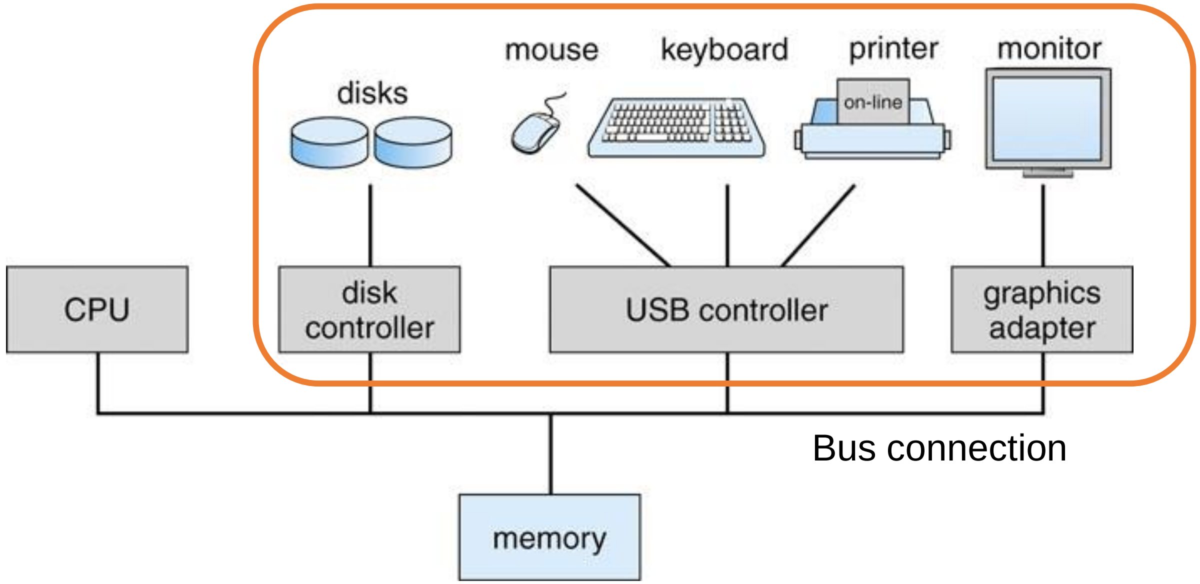
- application programs
- OS, system programs

Hardware

- CPU, memory (storage), battery
- input/output devices

Images from Techstereo

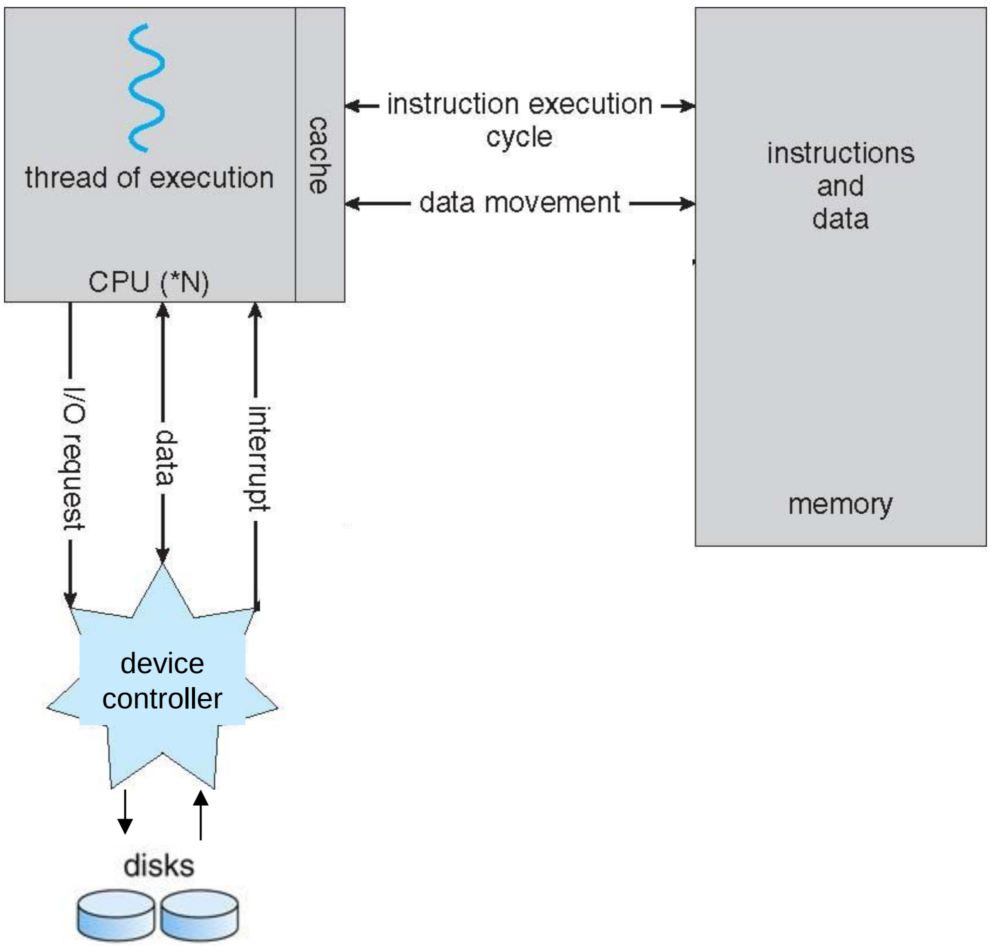
Computer system hardware



Inputs/outputs

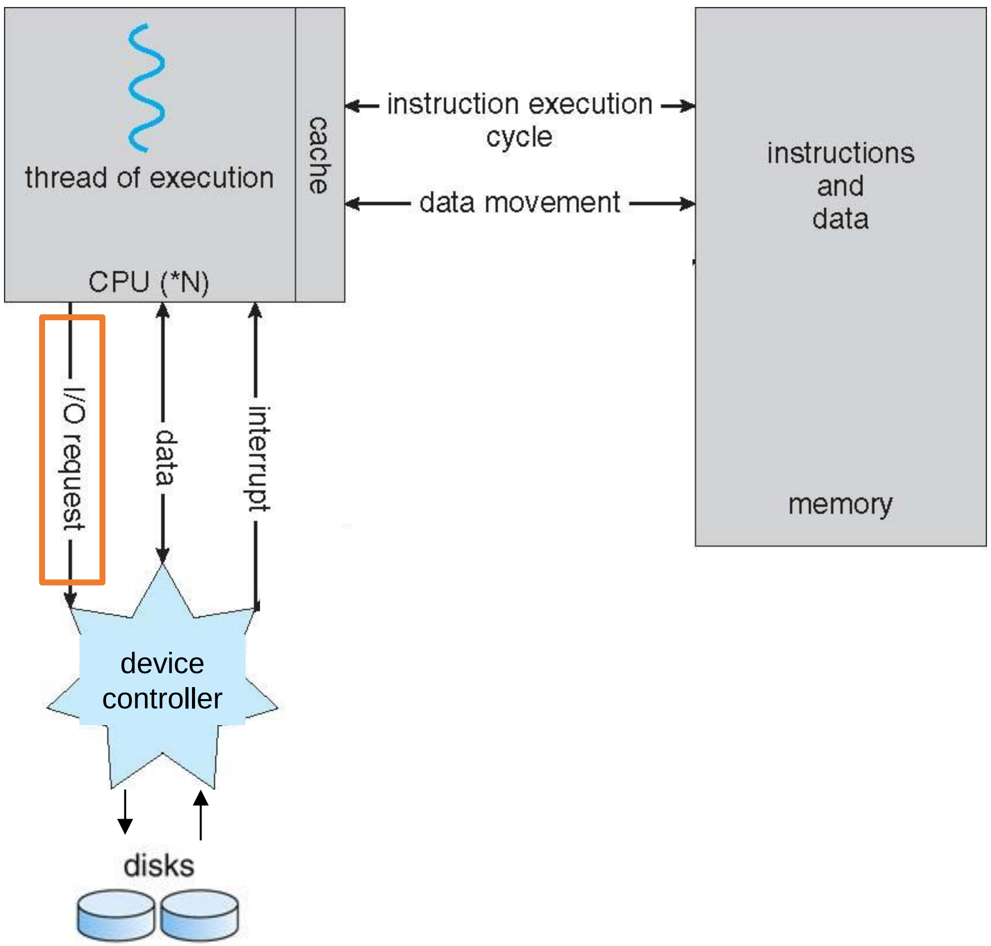
Bus connection

How does a computer work?



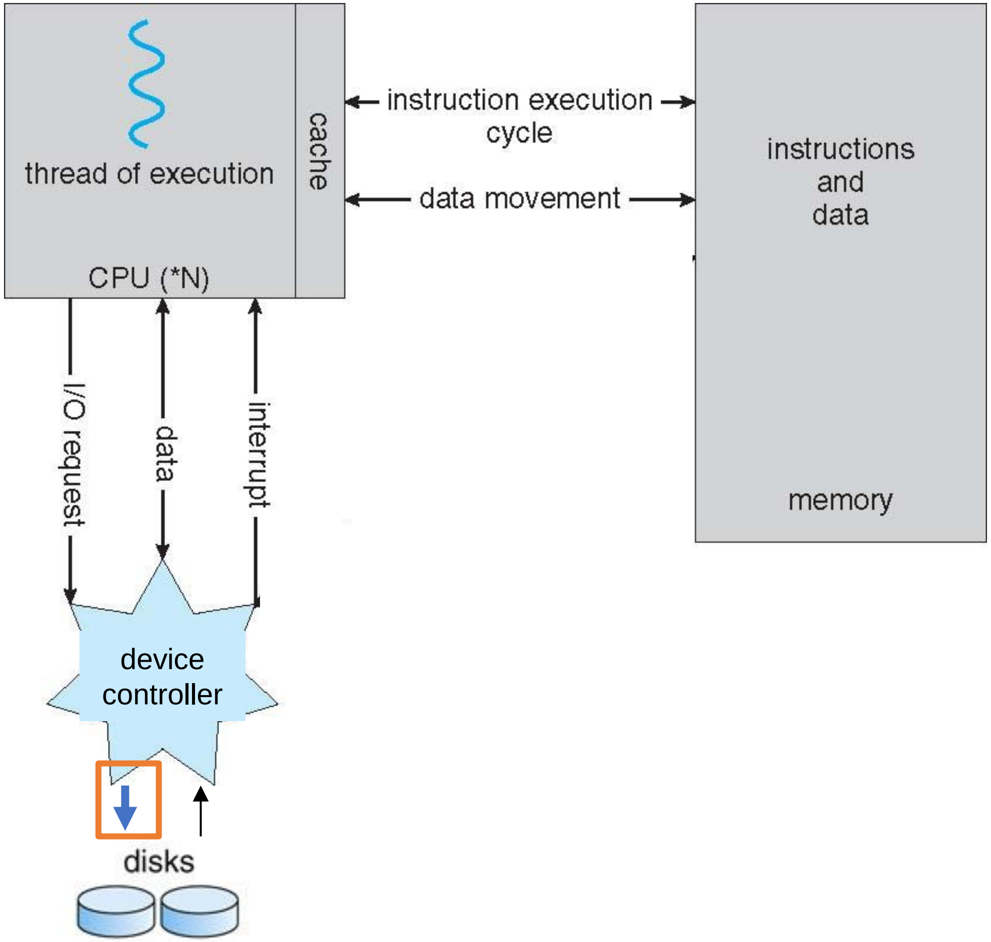
- **Example: access data on a hard drive:**

How does a computer work?



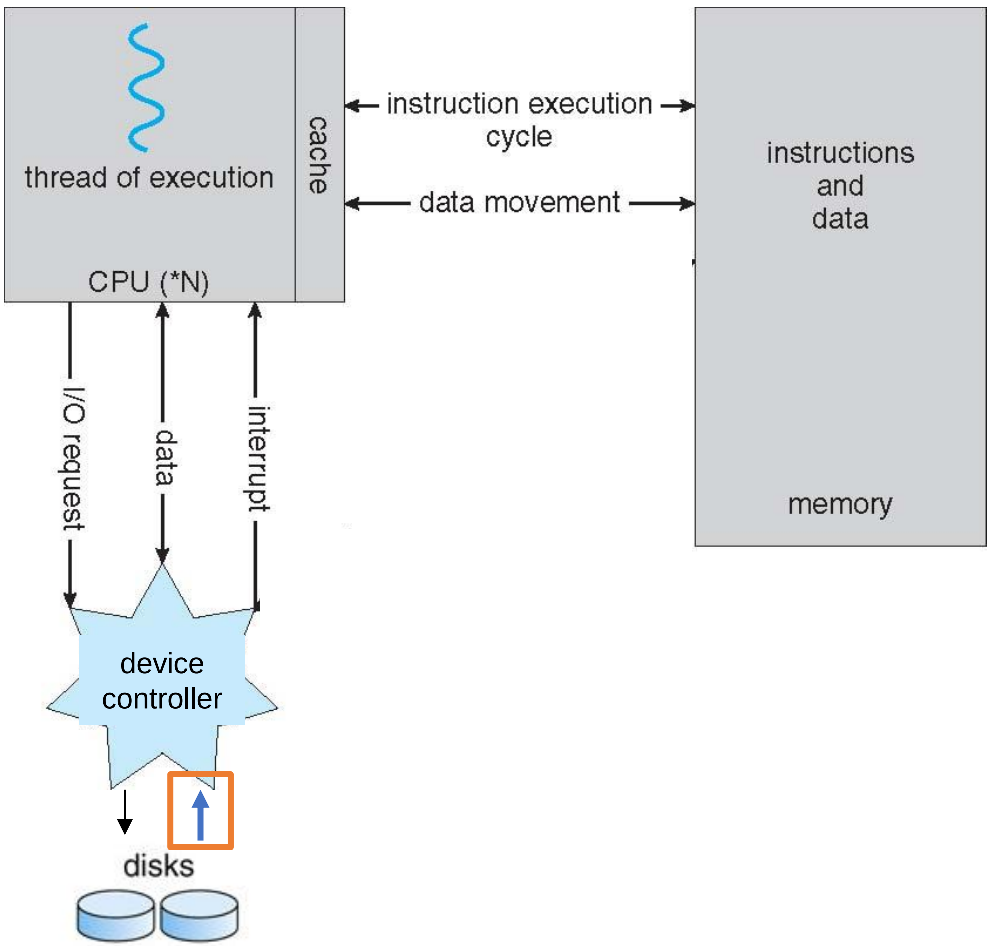
- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)

How does a computer work?



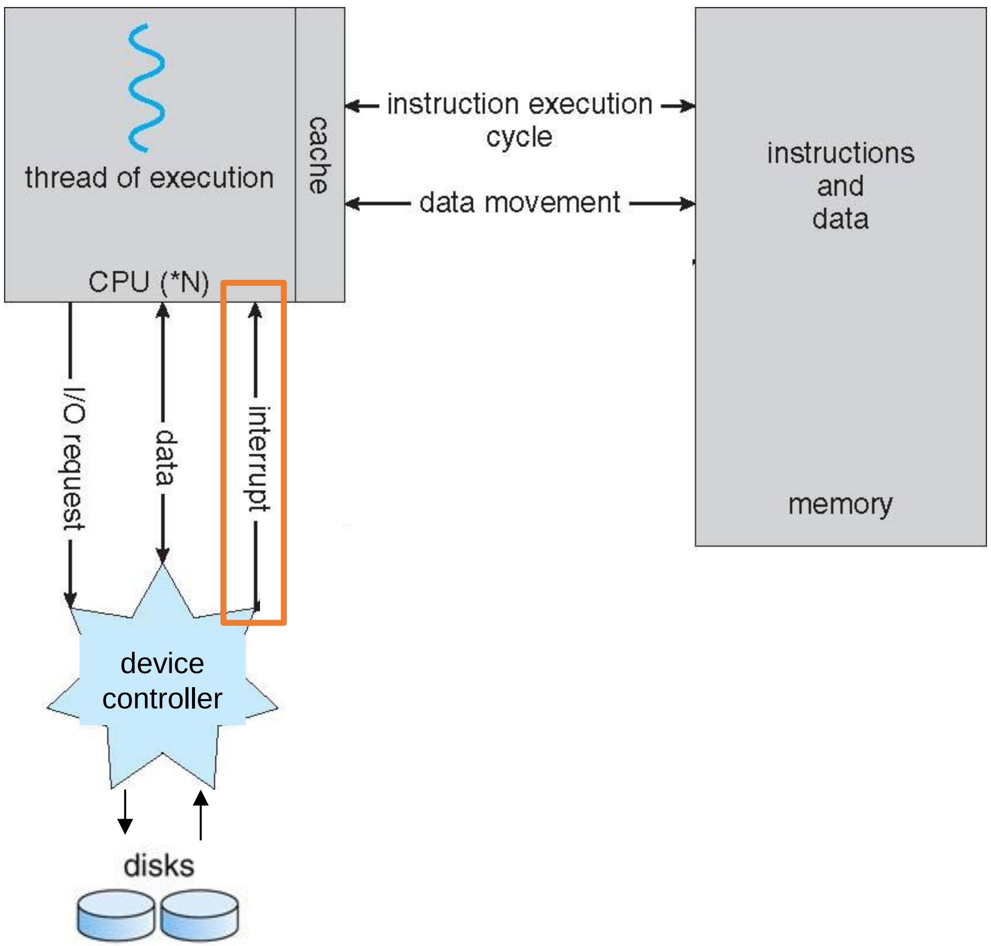
- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**

How does a computer work?



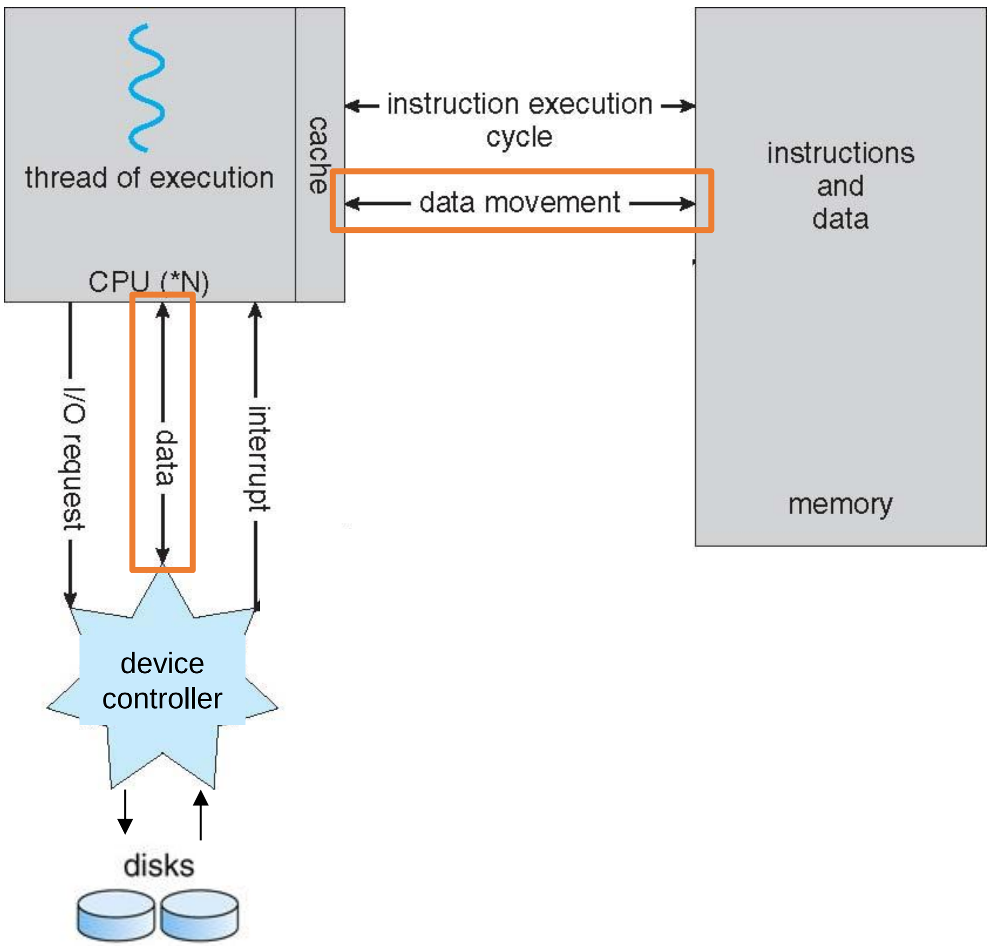
- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**
  3. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***

How does a computer work?



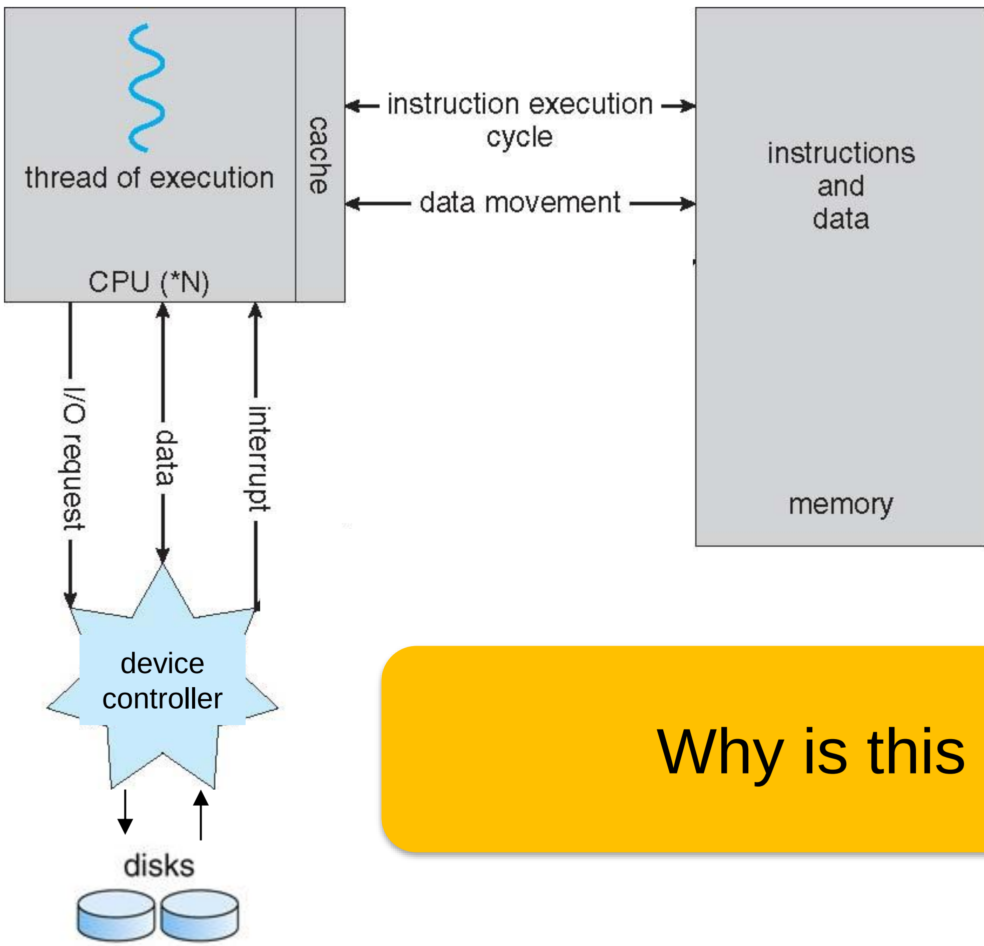
- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**
  3. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***
  4. **Device controller tells the CPU that the data is ready** using an ***interrupt***

How does a computer work?



- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**
  3. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***
  4. **Device controller tells the CPU that the data is ready** using an ***interrupt***
  5. **CPU reads data** in the device controller and **copies it in memory**

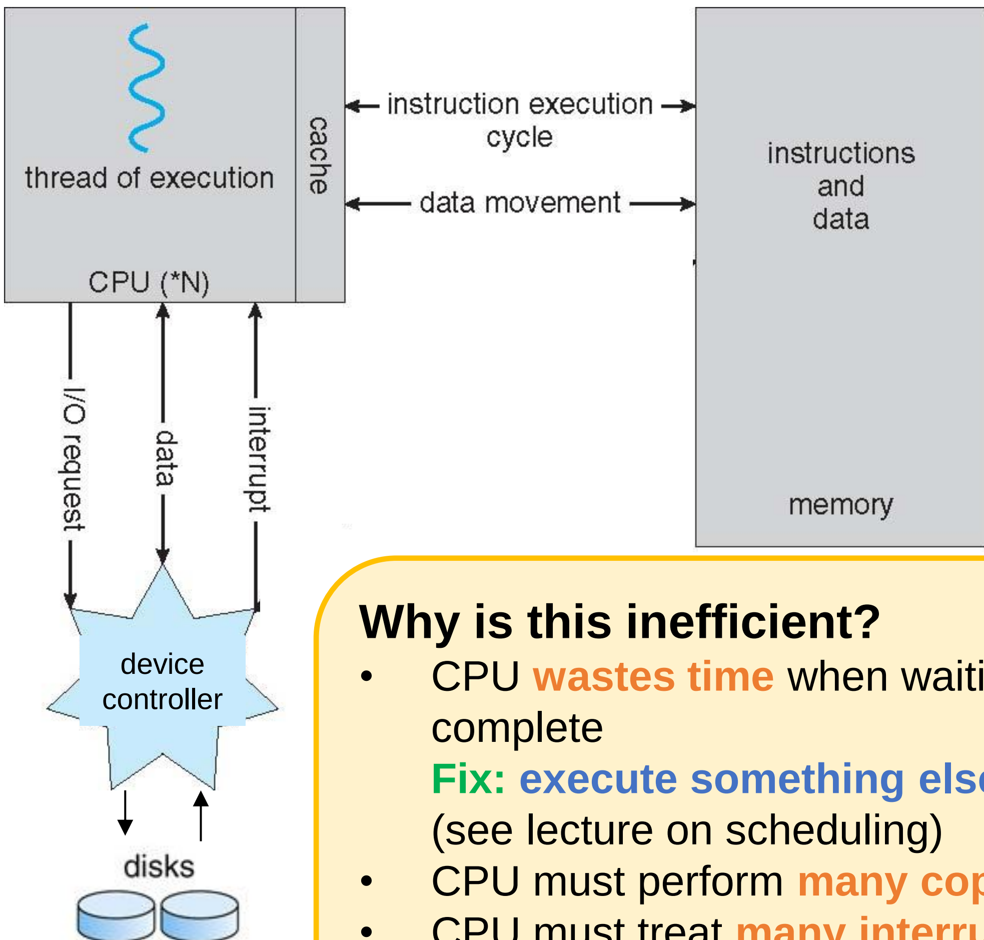
How does a computer work?



- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**
  3. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***
  4. **Device controller tells the CPU that the data is ready** using an ***interrupt***
  5. **CPU reads data** in the device controller and **copies it in memory**
  6. Return to 1 to read next data

Why is this inefficient?

How does a computer work?



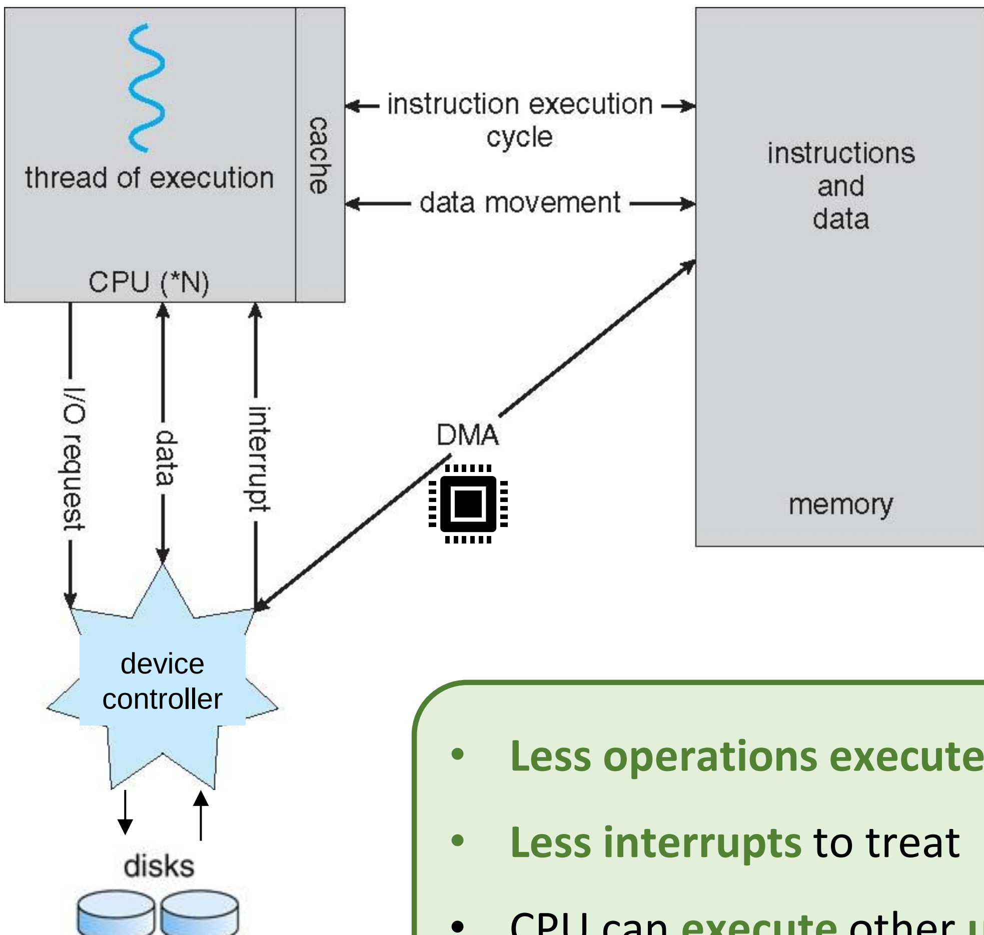
- **Example: access data on a hard drive:**
  1. **CPU tells the device controller what data it wants to read** (CPU writes command + address in ***device controller registers***)
  2. **Device controller accesses the hard drive**
  3. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***
  4. **Device controller tells the CPU that the data is ready** using an ***interrupt***
  5. **CPU reads data** in the device controller and **copies it in memory**
  6. Return to 1 to read next data

Why is this inefficient?

- CPU **wastes time** when waiting for the I/O operation to complete
  Fix: execute something else on the CPU while waiting (see lecture on scheduling)
- CPU must perform **many copy operations**
- CPU must treat **many interrupts**
  Fix: use direct memory access (DMA)

Imagine we must move 1000 data

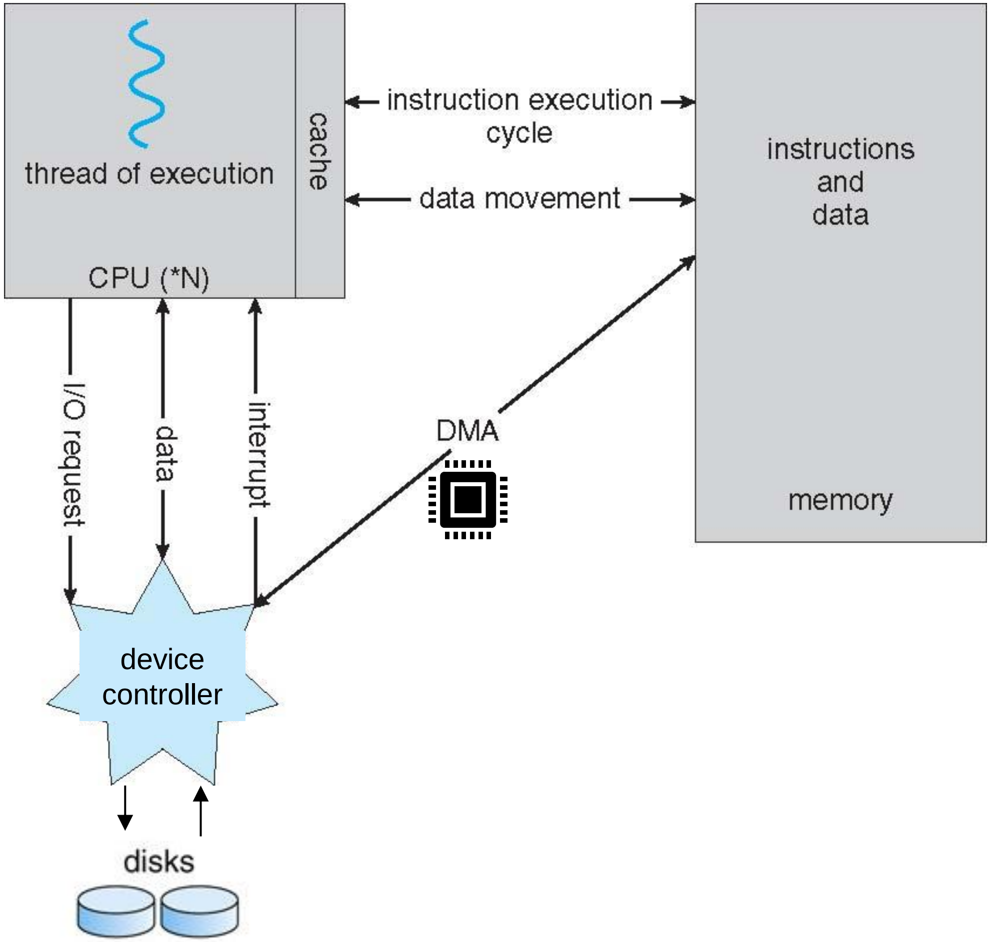
How does a computer work?



- **Example: access data on a hard drive using a DMA:**
  1. **CPU tells the DMA what block of data it wants to read** (CPU writes command + **start address** + **size** of block of data it wants to read in *DMA registers*)
  2. **DMA configure the I/O device controller for every data that must be red**
  3. **Device controller accesses the hard drive**
  4. **Hard drive recovers data** on disc and writes it in the ***device controller buffer***
  5. **DMA reads data** in the device controller and **copies it in memory**
  6. **Once all data are transferred** in the memory, **DMA tells the CPU the I/O transfer is completed using an interrupt**

- **Less operations executed** by the CPU
- **Less interrupts** to treat
- CPU can **execute** other **useful code in parallel**

How does a computer work?



Someone must **decide what to execute** while waiting for the I/O

Someone must provide code to **configure** the HW devices and **treat** the interrupts

Someone must ensure that the **program state remains consistent** when switching between executing tasks

Many concerns

- **HW operates at a very low abstraction level compared to user programs**
  - A mediator is needed in between

- **HW resources are shared among running programs (in time and in space)**
  - A burden that must be taken away from the programmer

- HW can be expensive
  - **Use of HW resources needs to be optimized**

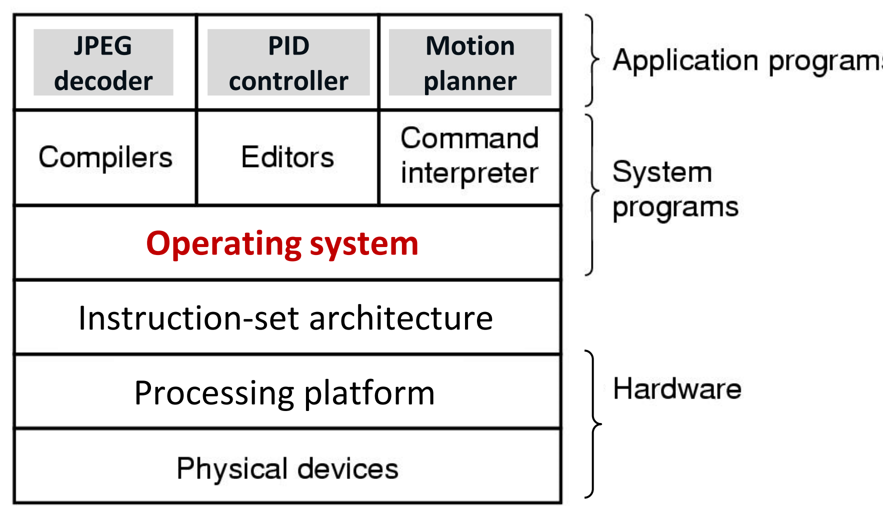
Why do we need an operating system?

An operating system is a piece of software that acts as an intermediary between **users/applications** and **computer hardware**.

It provides a level of abstraction that **hides the gory details of the HW architecture from applications**.

It **manages** the **sharing of HW resources** among applications and users

It **optimizes performance** through **resource management**



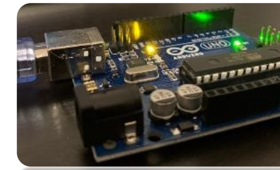
Without operating systems …

In a "bare-metal implementation", application developers must

- **Know hardware platform details** to be able to write applications
- **Manage the memory and address space** (very complex)
- **Manage I/O operations**
- **Manage communications**
- **Manage the file system**

Application code

Hardware

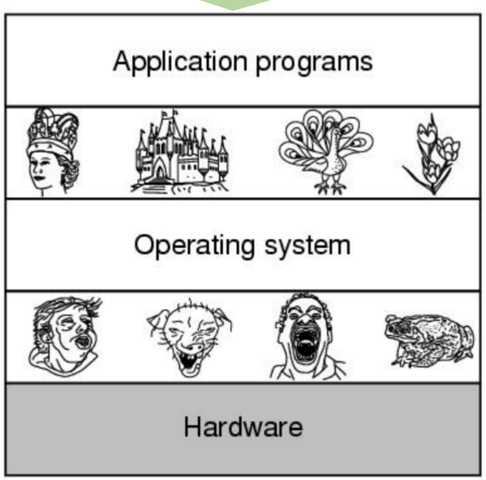


Application development becomes **time consuming** and **error prone**

Low portability (the code cannot be easily used on other HW platforms)

It also becomes very **hard to use third-party code and libraries**

Operating systems turn ugly (and **complex**) **hardware** into **beautiful abstractions**



Beautiful interface

Ugly (and complex) interface

Source of the figure: Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc.

What does an operating system provide?

Process management

- Creation and deletion
- Scheduling, suspension, and resumption
- Synchronization, inter-process communication

Memory management

- Allocate and deallocate memory space as requested
- Efficient utilization when the memory resource is heavily contended
- Keep track of which parts of memory are currently being used and by whom

I/O management

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices

File management

- Manipulation of files and directories
- Map files onto (nonvolatile) secondary storage -disks
- Free space management and storage allocation
- Disk scheduling

Error detection

- Debugging facilities

Protection

- concurrent processes should not interfere with each other (memory)

Security

- Against other processes and outsiders

Accounting

Using timers

- CPU cycles,
- main memory usage
- I/O devices usage
- …

What does an operating system provide?

Process management

- Creation and deletion
- Scheduling, suspension, and resumption
- Synchronization, inter-process communication

Memory management

- Allocate and deallocate memory space as requested
- Efficient utilization when the memory resource is heavily contended
- Keep track of which parts of memory are currently being used and by whom

I/O management

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices

File management

- Manipulation of files and directories
- Map files onto (nonvolatile) secondary storage -disks
- Free space management and storage allocation
- Disk scheduling

Error detection

- Debugging facilities

Protection

- concurrent processes should not interfere with each other (memory)

Security

- Against other processes and outsiders

Accounting

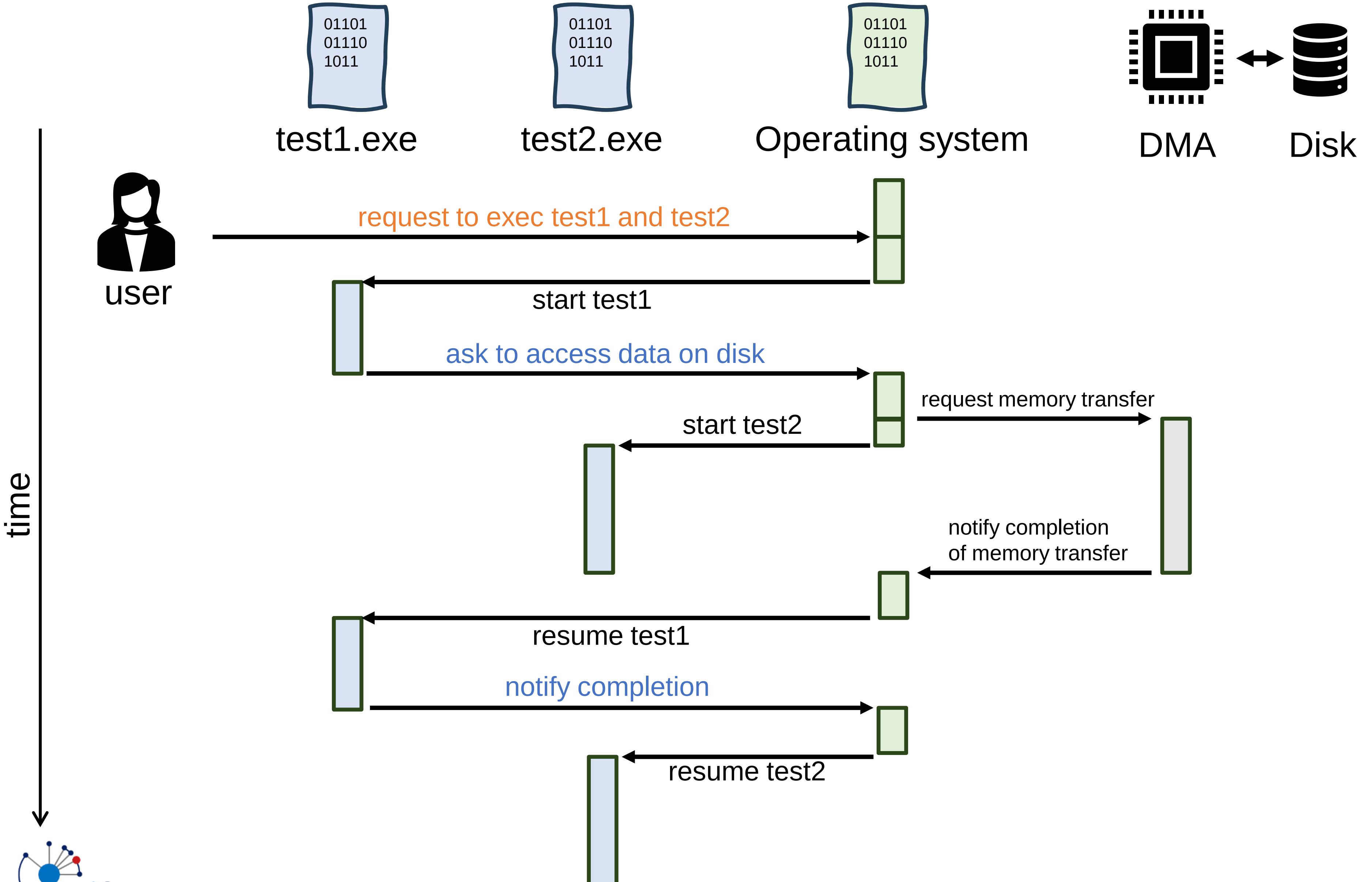
Using timers

- CPU cycles,
- main memory usage
- I/O devices usage
- …

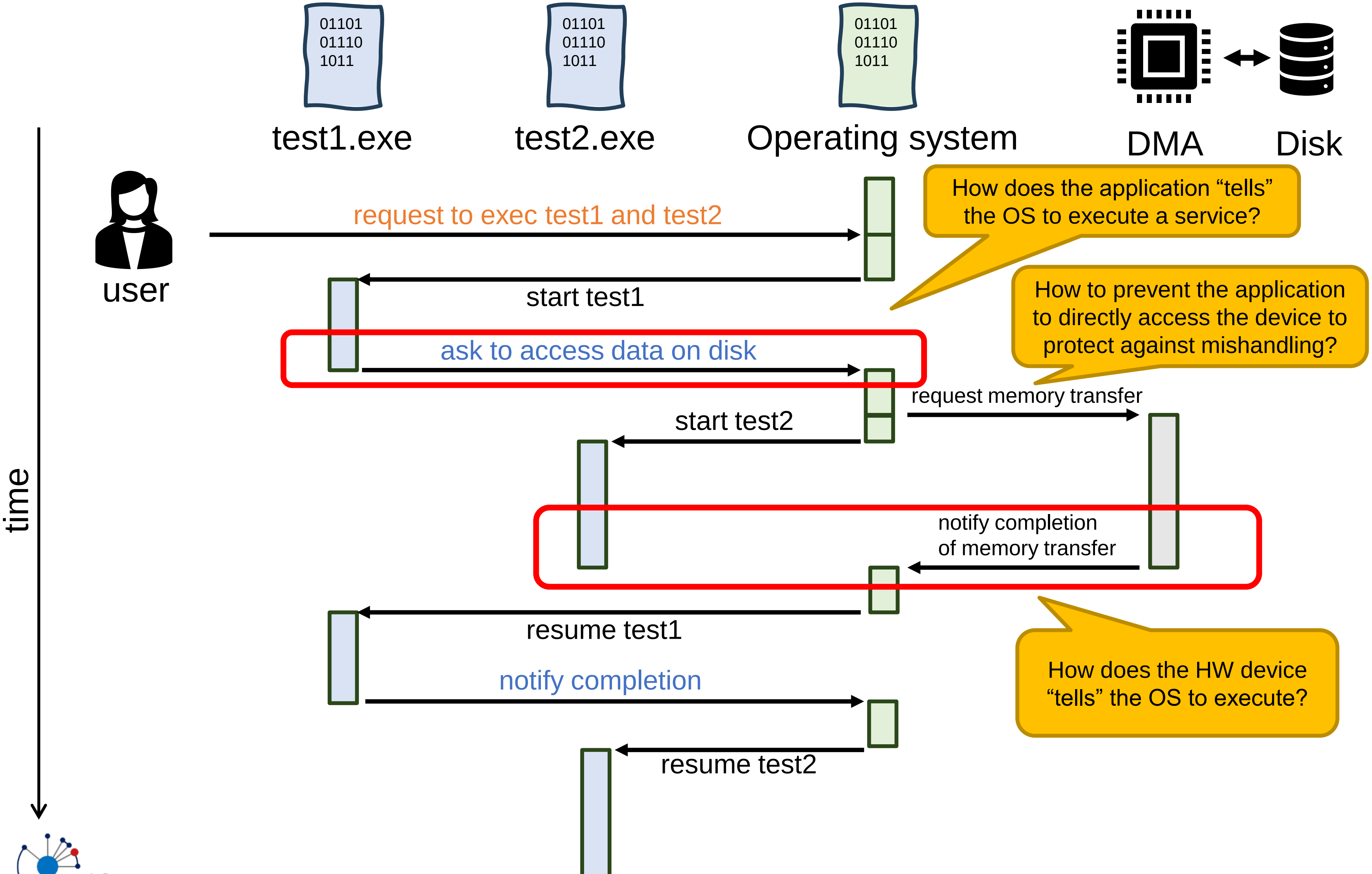
Covered in this course

How applications/external devices interact with an OS?

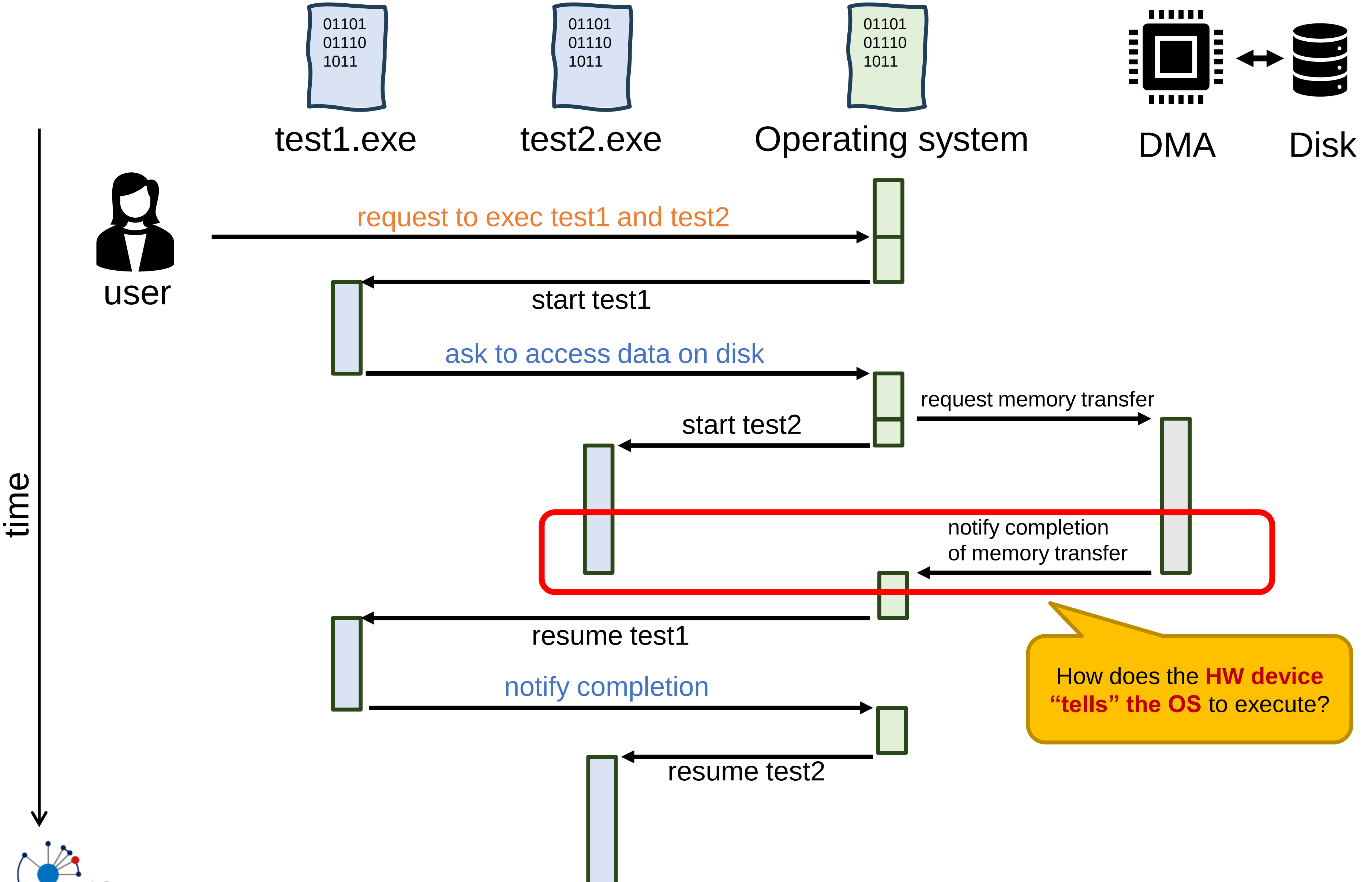
How OS and application/devices interact?



How OS and application/devices interact?

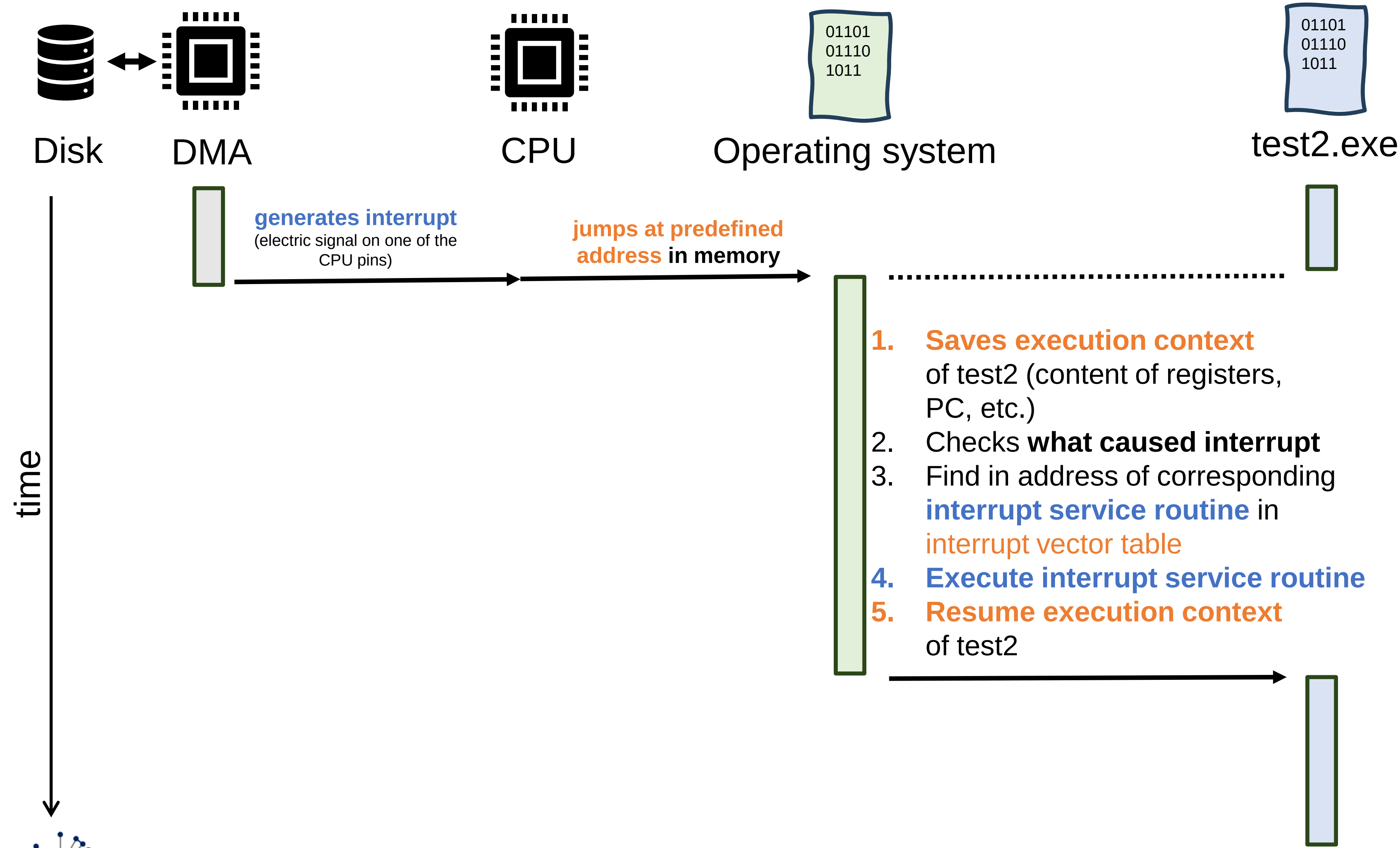


How OS and application/devices interact?

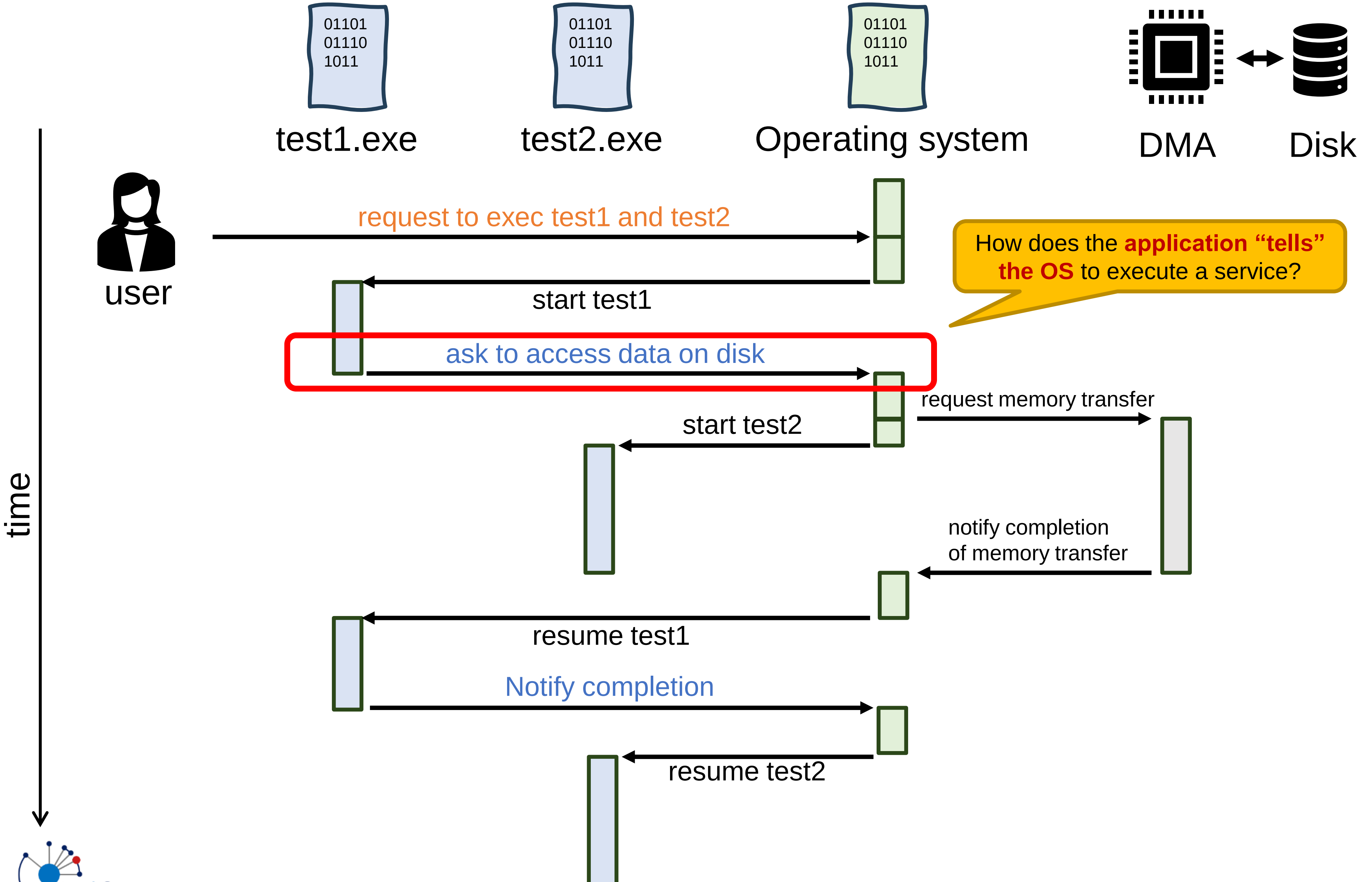


Communication with devices: OSs are interrupt driven

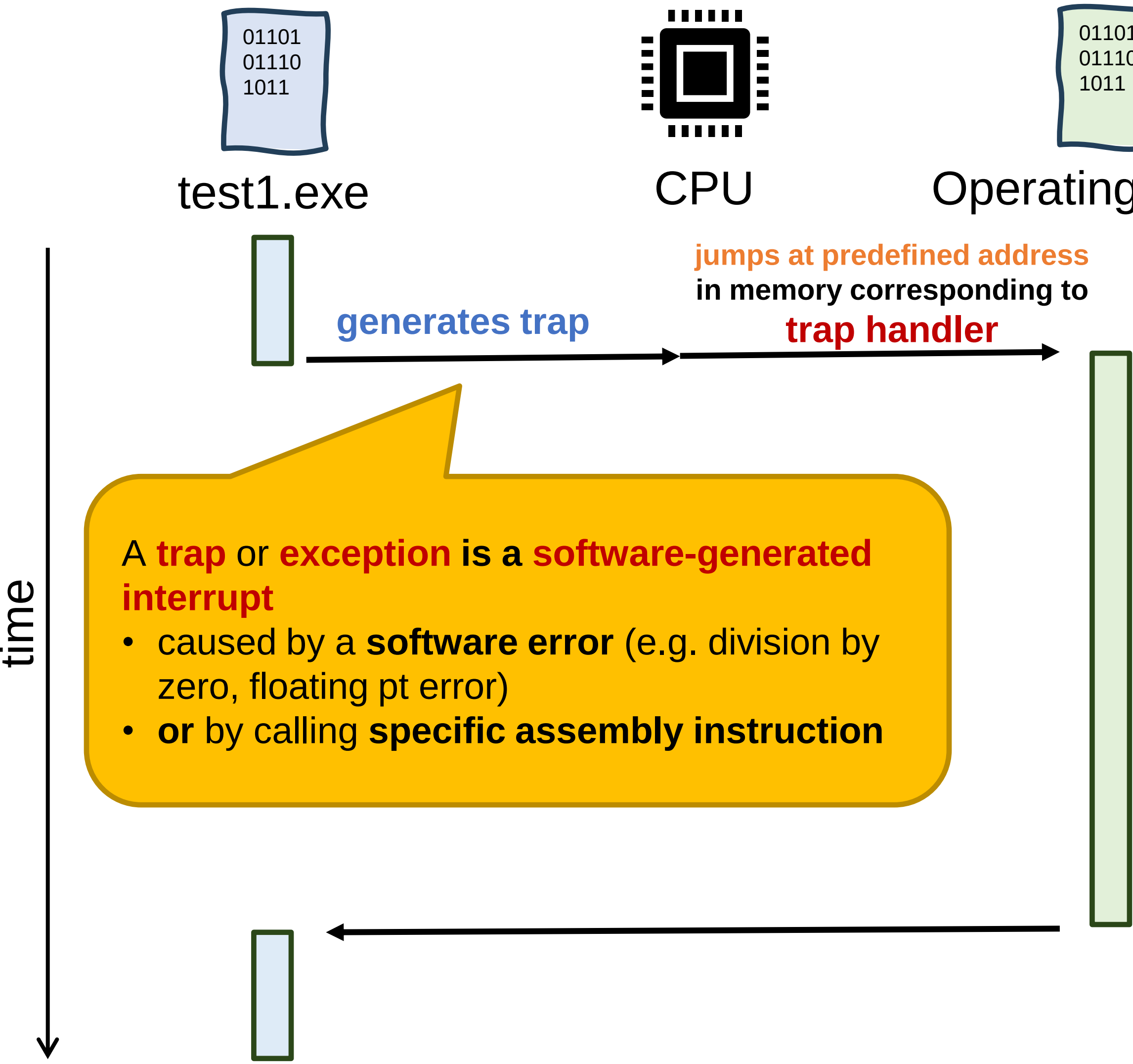
Communication with devices: OSs are interrupt driven



How OS and application/devices interact?

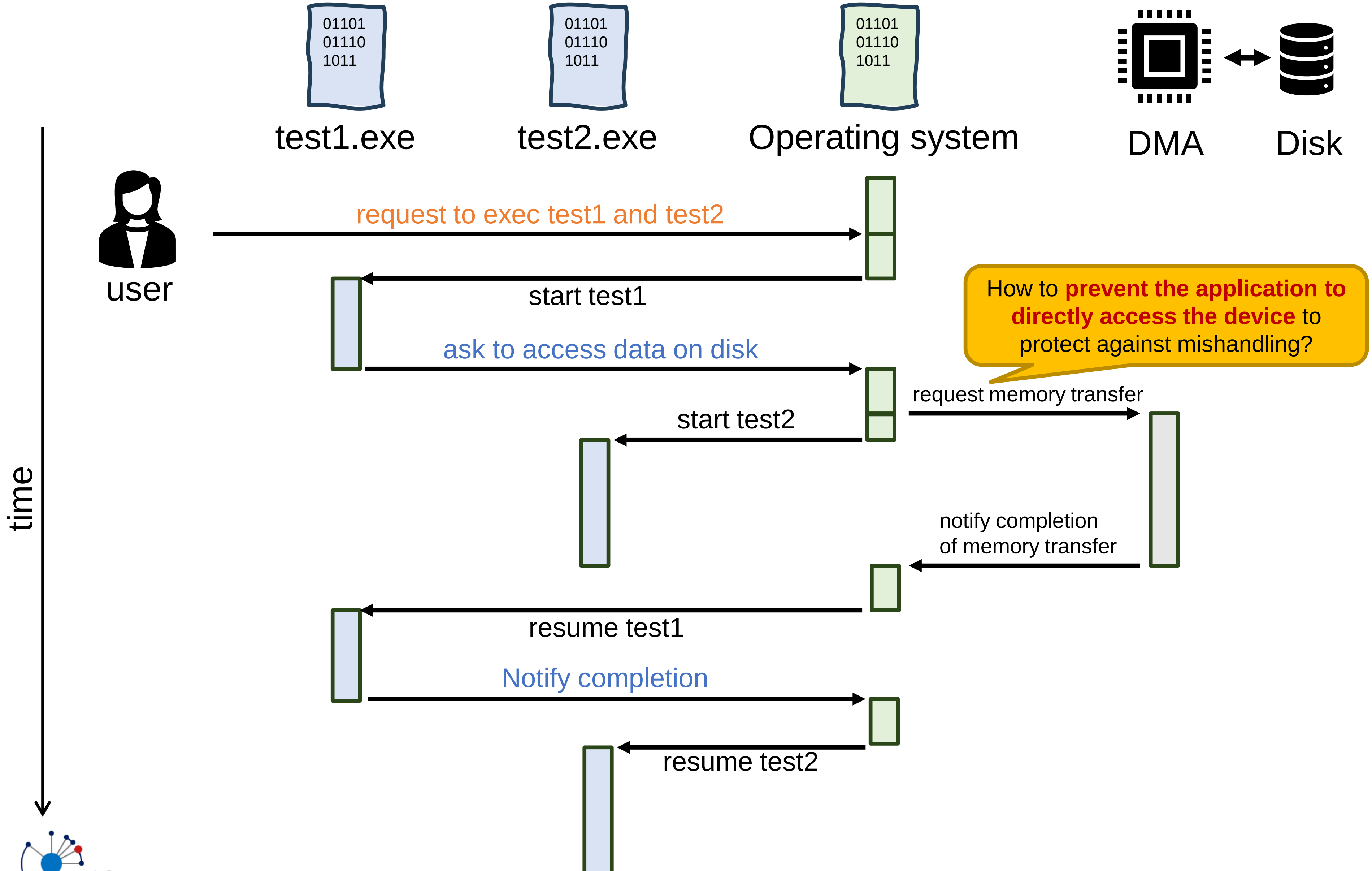


Communication between applications and OS: OS’s are interrupt driven



1. **Saves execution context** of test1 (content of registers, PC, etc.)
2. Checks **why trap was generated**
3. Execute **requested service**
4. Resume execution context of test1

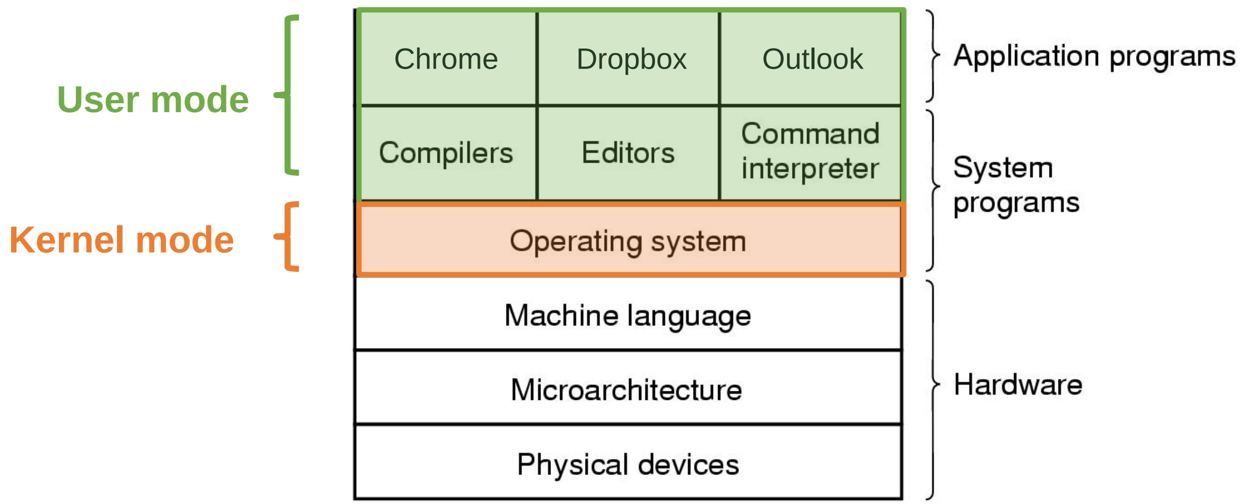
How OS and application/devices interact?



OS: Dual-mode operation

OS: Dual-mode operation

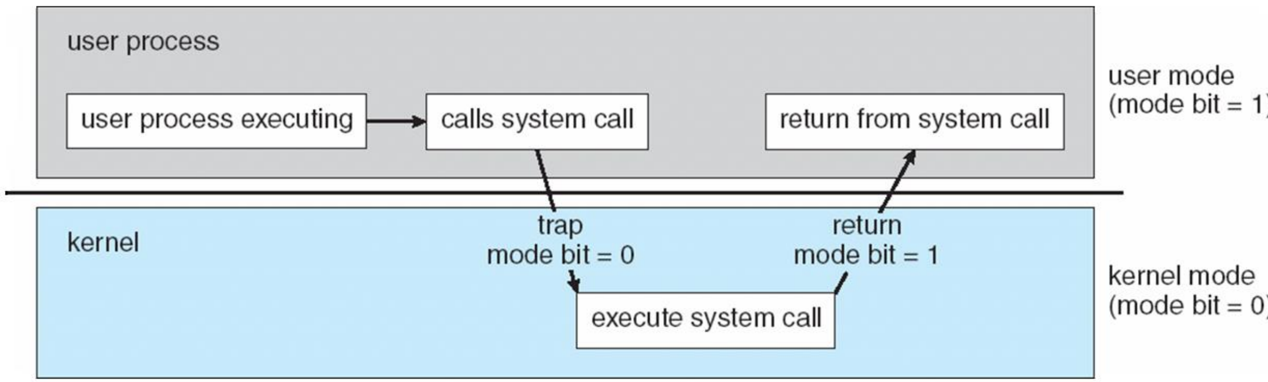
- **Different privileges** required **for different types of code**
  - **User code** is executed in **user mode**
  - **Operating system** is executed in **kernel mode**
  - Most instructions can be executed in **user mode**, but not all
    - For instance, in user mode, the CPU cannot access memory locations reserved for OS
  - Some “privileged” instructions are only executable in **kernel mode**



Dual-mode allows the OS to ***protect*** **itself** and **other system components**

OS: Dual-mode operation

- **Mode bit** provided by hardware
  - Provides ability to **distinguish** when system is **running user code or kernel code**
  - **A *system call* changes the execution mode** to kernel mode, returning from the system call resets the mode to user mode



Example: read data from file using *read(fd, buffer, nbytes)*

Reads *nbytes* bytes from *fd*, storing it in *buffer:*

1. push parameters needed by *'read(.,.,.)'* (pointer to fd, buffer address, number of bytes) on the stack, in registers or in pre-defined place in memory write a code identifying *'read(.,.,.)'* in a register so that the trap handler knows that it must call the code for *read*
2. generate a trap: switch mode & call trap handler
3. Trap handler calls and execute the *read(.,.,.)* function handler
4. Return in user mode and give control back to the caller

Example:
read data from file using *read(fd, buffer, nbytes)*

Reads *nbytes* bytes from *fd*, storing it in *buffer:*

User mode

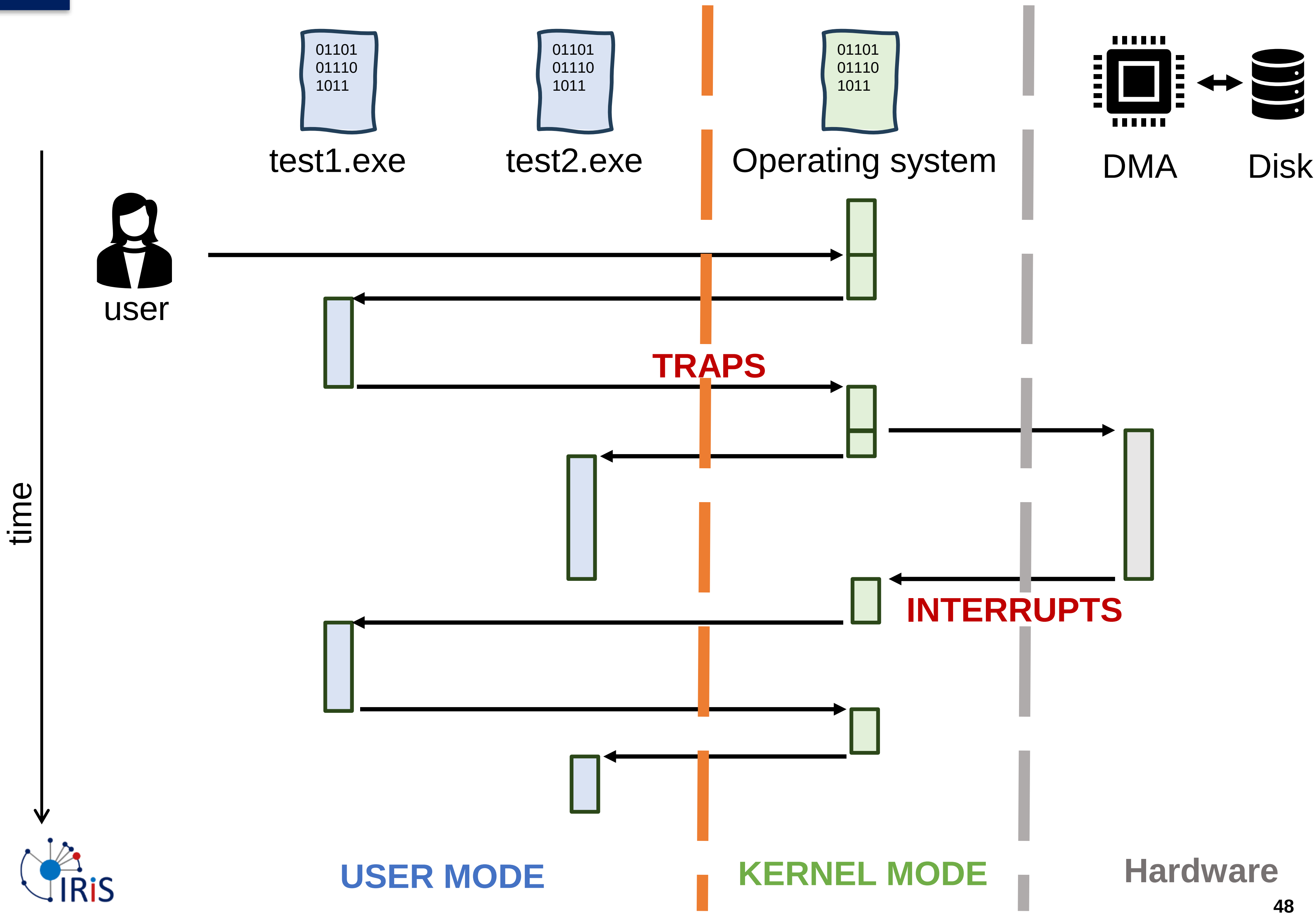
1. push parameters needed by *'read(.,.,.)'* (pointer to fd, buffer address, number of bytes) on the stack, in registers or in pre-defined place in memory
2. write a code identifying *'read(.,.,.)'* in a register so that the trap handler knows that it must call the code for *read*
3. generate a trap: switch mode & call trap handler

Kernel mode

4. Trap handler calls and execute *read(.,.,.)* function handler
5. Return in user mode and give control back to the caller

User mode

How OS and application/devices interact?



Agenda

- Course overview
- OS: place in the computer system
- Motivation & OS tasks
- Extra-functional requirements

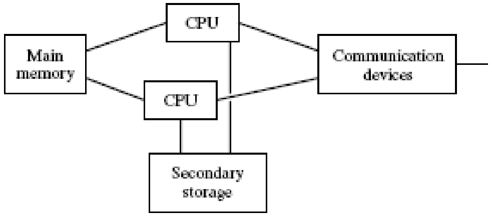
OS Motivation

- Deal with diversity
- Transparency
- Virtualization
- Support for shared functionality
- Portability
- Support for concurrency

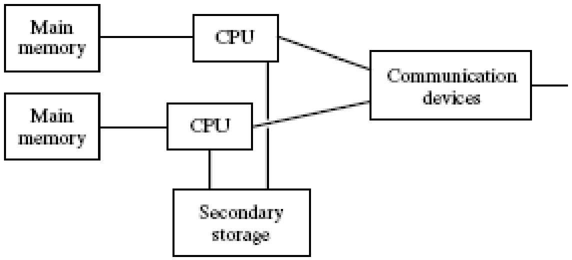
OS Motivation: Deal with diversity

- Many system implementations
  - **diverse CPUs** (Instruction Set Architectures)

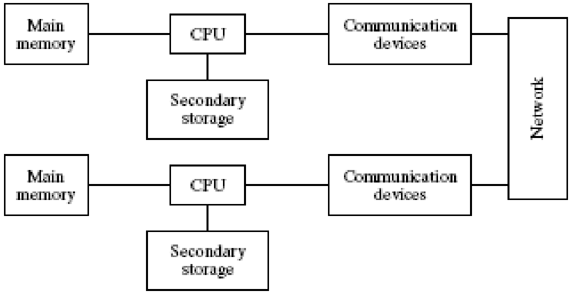
  - **diverse architectures and organization**



Multicore system



Clustered system

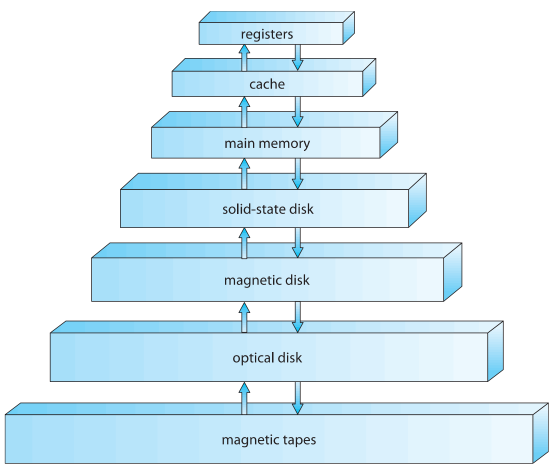


Distributed system

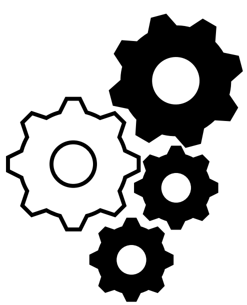
OS Motivation: Transparency

- **Hide details** with respect to a given issue.

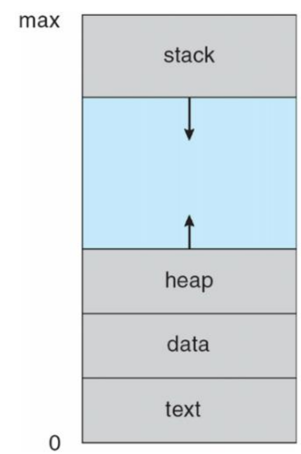
- Examples:
  - **processor architecture** (Instruction Set Architecture - ISA)
    - mechanism: use compiler
  - physical **memory size**
    - mechanism: **virtual memory** i.e., present linear memory model that is larger than physical
  - **Location** of programme and data **in physical memory**
    - mechanism: **indirection using logical memory address**

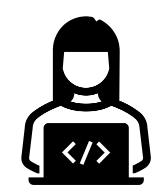


Reality



OS layer



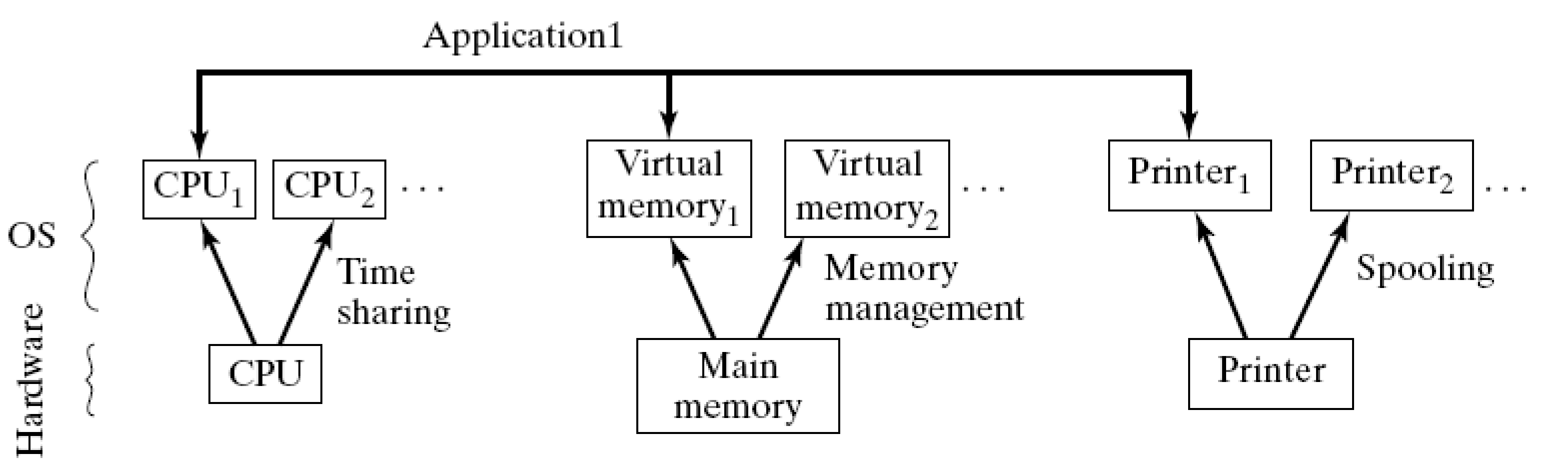


What the programmer sees

OS Motivation: Virtualization

- Provides a **simple, abstract, logical model** **of the system**
  - virtual memory, virtual CPU, virtual disk
  - **Program can be developed as if they were the sole user of the HW resources**

- Current systems tend to virtualize the entire hardware
  - i.e., it yields a virtual machine



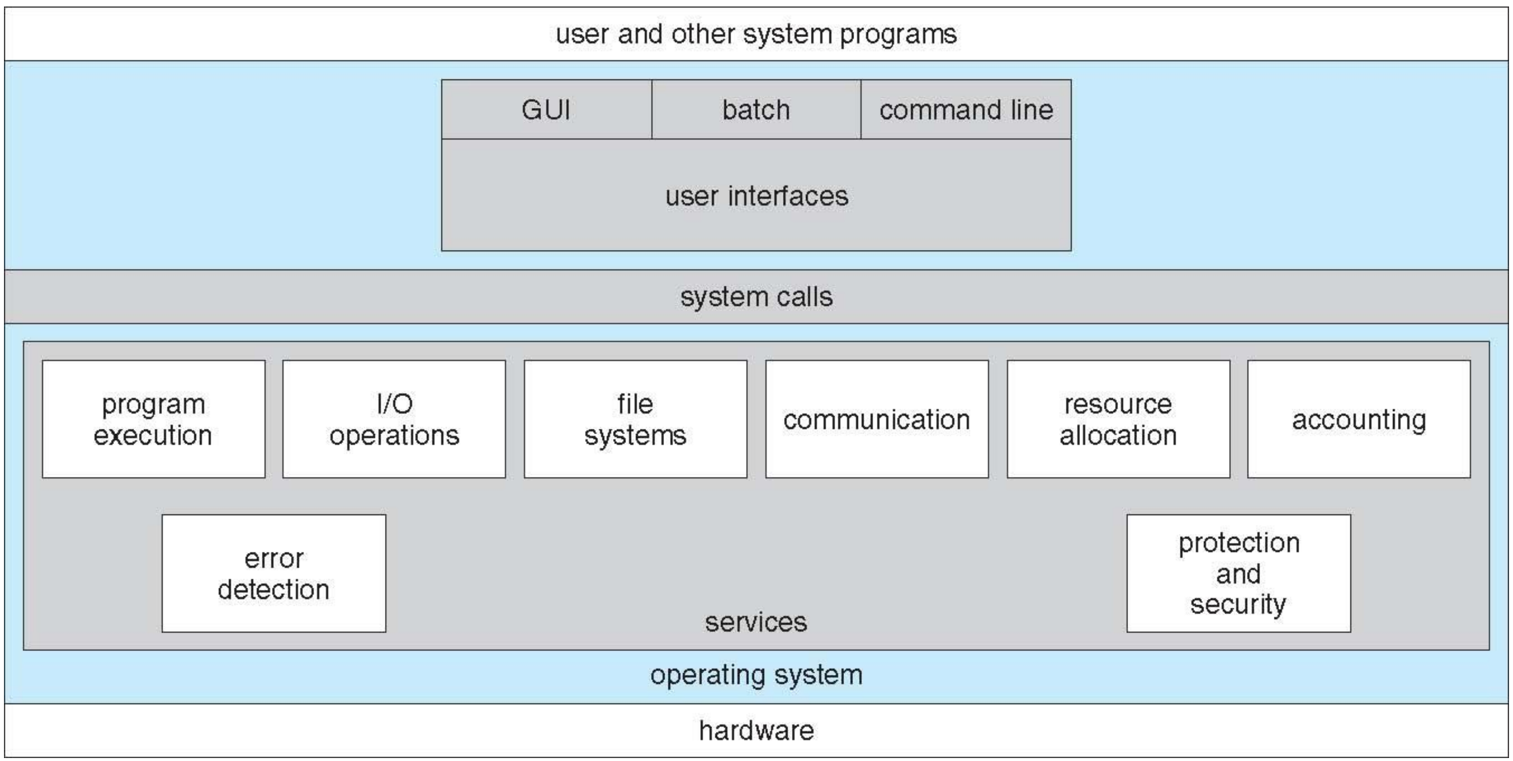
OS Motivation: Support for shared functionality

- OS task: provide *functionality common to most programs*
  - introduce well-defined ***abstractions*** of *concepts*
    - files and file systems instead of disk blocks
    - read from or write to I/O devices as if writing in a file instead of manipulating registers in device controller
    - exceptions and traps rather than ‘something goes wrong’
    - linear memory rather than memory blocks, pages and disk space

  - provide **system calls** to access the functions provided by the OS

OS Motivation: Support for shared functionality

- **System calls**: Programming interface to the services provided by the OS,…
- …typically written in a high-level language (C or C++)



OS Motivation: Support for shared functionality

- **Types of system calls:**
  - **Process management**
    - create, destroy, communication, synchronization, ...
  - **File management**
    - open, close, read, write, ...
  - **Memory management**
    - allocation, free, share memory between processes, ...
  - **Device management**
    - access control, open, attach, send/receive, ...
  - **Communications**
    - setup communications, exchange messages, ...
  - **Miscellaneous**
    - timers, inspect system resources, ...

OS Motivation: Support for shared functionality API vs. System calls

- System calls are mostly accessed by programs via a high-level **Application Programming Interface (API)**
  - Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X),
  - Java API for the Java Virtual Machine (JVM),
  - Cocoa Touch for iOS, Java based Android API for Android (runs on top of Android Runtime – previously on top of Dalvik VM).

Advantage of an API over system calls

- API works at a higher abstraction closer to programs need➔ **easier** to work with
- API calls may combine many system calls to implement higher level tasks (e.g., copy a file) ➔ **less calls** needed
- a program written using an API can compile/run on any system supporting the API ➔ **portable**

Disadvantage of an API to system calls

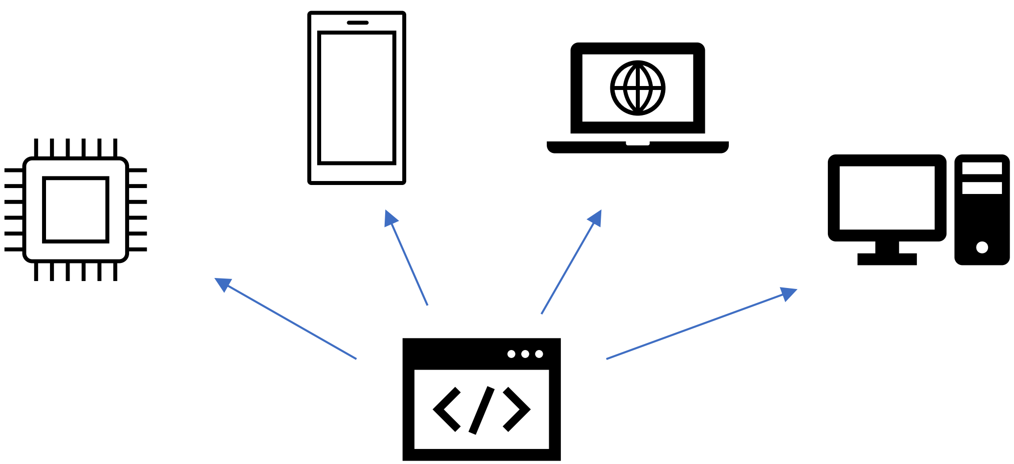
- Adds one more layer ➔ increased overhead

OS Motivation: Portability

- a program written using an OS or an API can compile/run on any system supporting that OS or API

- Protect investments in application software
  - Reduces cost to support several platforms
  - Reduces cost to upgrade/change the execution platform
  - Simplifies integration of several application components

- OS gives a ***unified machine view*** to applications, effectively defining a ***virtual machine***



OS Motivation: Support for concurrency

- **The machine must be *shared*** between multiple activities (‘tasks’, ‘processes’) and multiple users

- OS:
  - realizes ***concurrency transparency*** (virtualization)
    - each task virtually has a (virtual) machine of its own
  - ***manages and protects*** tasks from each others
    - Limits resource usage between tasks *and* between users
    - e.g., processor, memory, files, i/o equipment, ...
  - **ensures *efficiency*** by scheduling, e.g.,
    - to avoid starvation;
    - to enforce timing requirements;
    - to reduce power consumption;
    - ...

Agenda

- Course overview
- OS: place in the computer system
- Motivation and OS tasks
- Extra-functional requirements

OS extra-functional requirements

- **Extensibility**
  - support for adding application-specific (domain-specific) functionalities
- **Scalability**
  - wide range of environments, functionalities, machines
  - Scale to large-enough number of tasks, cpus, clusters, …
- **Dependability**
  - robust, correct, safe and secure
  - How dependable? → depends… on the application domain

Different flavors of operating systems

- **Real-time OS**
  - predictability
    - *known* performance instead of *high* performance (all resources)
    - Known bounds on the maximum latencies (response times) of API calls
  - support for dealing with real-time control
    - predictable scheduling policies
    - explicit control over resources
    - real-time facilities: clocks and timers
  - stringent dependability

- **Embedded OS**
  - small footprint (e.g., leave all superfluous parts out)
  - low system requirements (e.g. processor speed, energy)
  - stringent dependability

Popular real-time operating systems (RTOS)

MOST POPULAR REAL-TIME OPERATING SYSTEMS (2020)

- Deos (DDC-I)
- embOS (SEGGER)
- FreeRTOS (owned by Amazon since 2017)
- Integrity (Green Hills Software)
- Keil RTX (ARM)
- LynxOS (Lynx Software Technologies)
- MQX (Philips NXP / Freescale)
- Nucleus (Mentor Graphics)
- Neutrino (BlackBerry)
- PikeOS (Sysgo)
- SafeRTOS (Wittenstein)
- ThreadX (Microsoft Express Logic)
- µC/OS (Micrium)
- VxWorks (Wind River)
- Zephyr (Linux Foundation)

Compliant with the **safety standard** DO-178 required in **avionics** (spin-out from Honeywell)

Used in automotive industry (it is free for non-commercial use)

It is the second most popular/used operating system for embedded systems (according to a study by www.embedded.com, 2019)

Only supports ARM-Cortex M

It is also the kernel of QNX

Small company that is now part of Thales

Priority-based preemptive real-time kernel for microcontrollers

microkernel

Source: https://www.embedded.com/2019-embedded-markets-study-reflects-emerging-technologies-continued-c-c-dominance/

Most Used	World	Americas	Europe	Asia
Embedded Linux	31%	32%	31%	26%
FreeRTOS	27%	25%	24%	37%
Android	14%	12%	10%	26%

Source (2020) RTOS: https://www.lynx.com/embedded-systems-learning-center/most-popular-real-time-operating-systems-rtos

Summary

- Operating system
  - is an **interface** between HW and applications
  - abstracts the HW by providing high level services
  - manages concurrent access to HW devices
  - optimizes efficiency

- The operating system protects itself and other components using two modes of executions (user vs kernel mode)

- **HW** devices **communicate** using **interrupts**
- **SW** applications **use** **system calls** to perform **privileged operations**

Summary - OS tasks

- ***Abstraction***
  - provide useful generic concepts/functionalities
  - .... to handle complexity (transparency)
- ***Virtualization***
  - provide the same abstract model for a wide range of systems
  - each **process/user sees single machine, linear memory, ...**
- ***Resource management***
  - resource sharing
  - optimize performance
  - access control for protection and security
  - accounting and usage limit enforcement

Next lecture

- **Friday at 8h45**

- **How to implement concurrency in an OS**
  - Quiz
  - Exercises

- **Prepare!**
  - **11 pages to read in the reference book**
  - **25 minutes of video**